Agile Software Development Practices

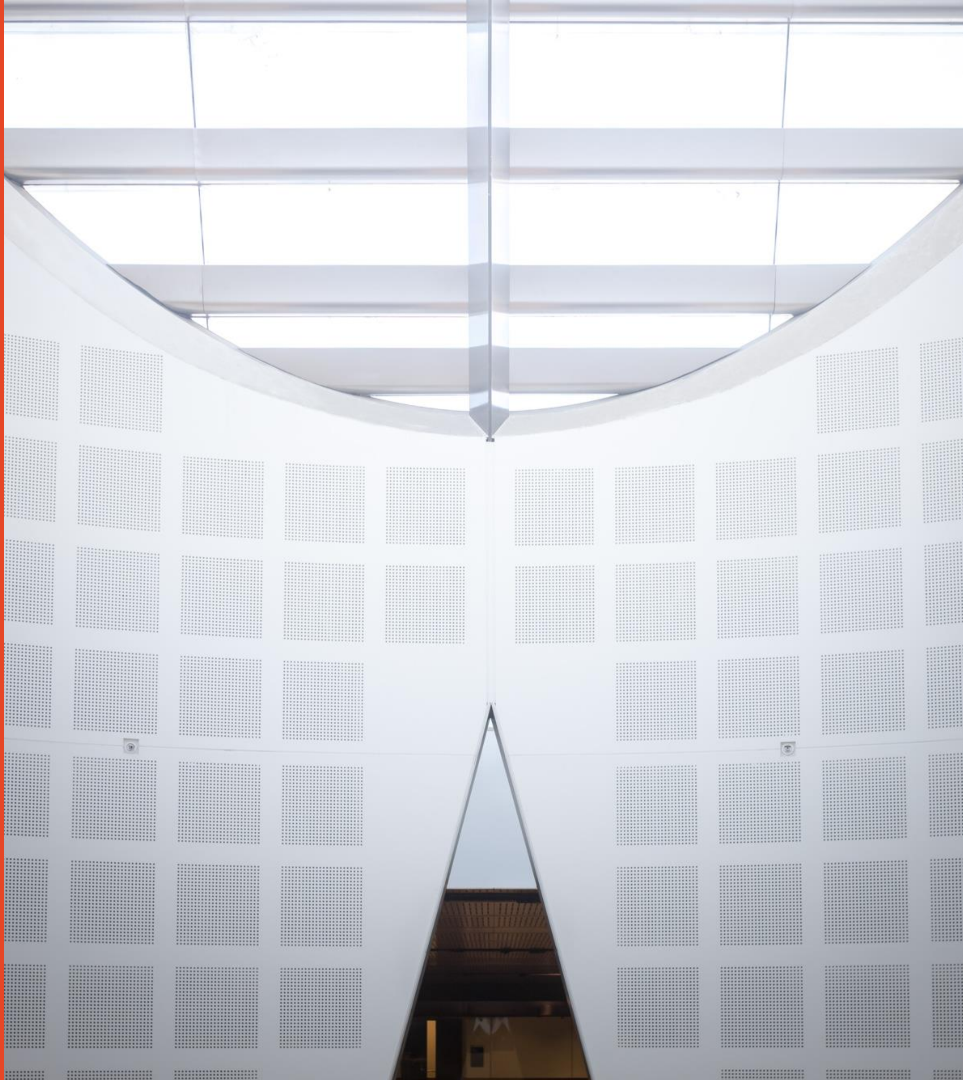
SOF2412 / COMP9412

System Build Automation

Dr. Basem Suleiman

School of Information Technologies

THE UNIVERSITY OF SYDNEY

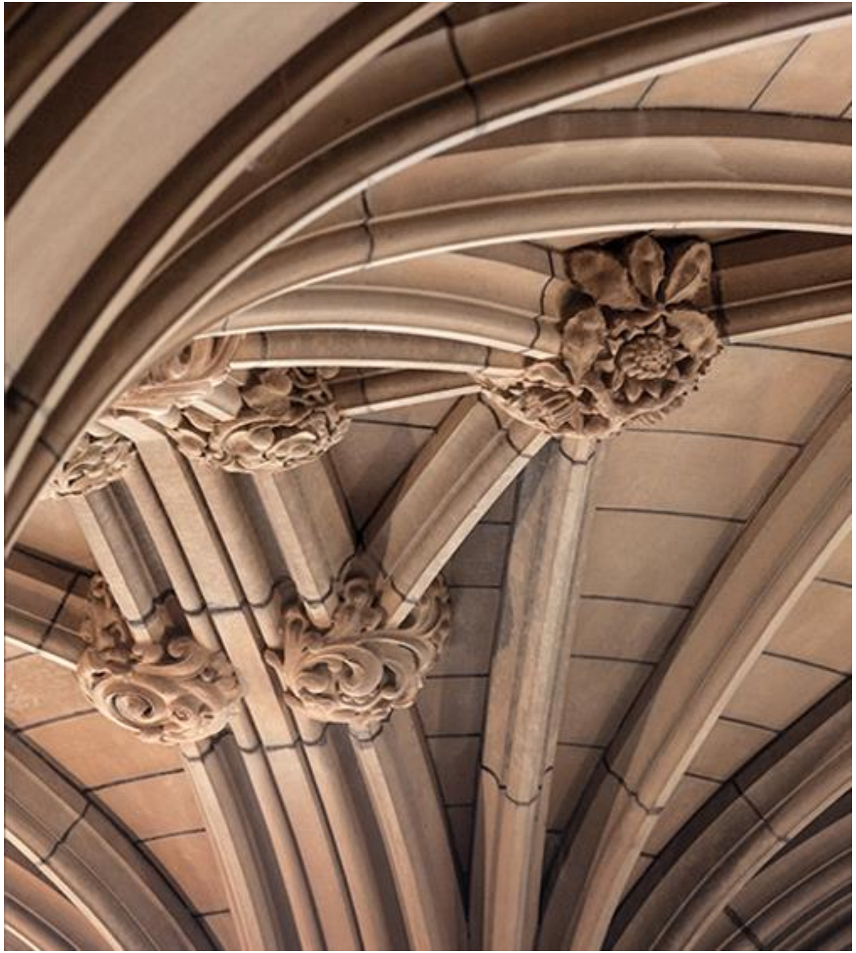


Agenda

- Software Configuration Management
  - 1 System Building
  - 2 Agile System Build

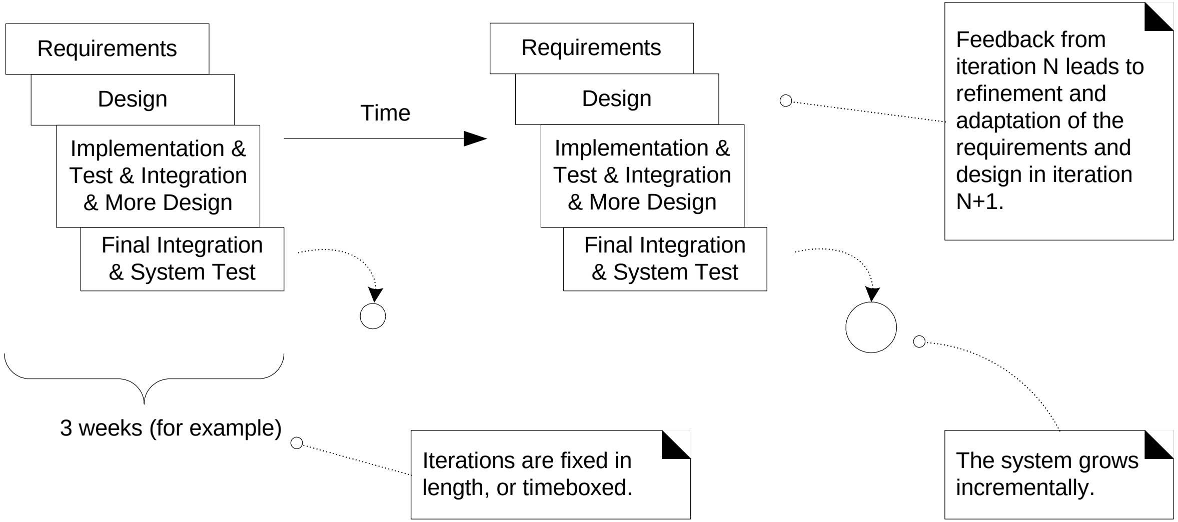
- Software Build Automation Tools
  - 1 Ant
  - 2 Maven
  - 3 Gradle

Software Configuration Management



Revisit – Software Development Process Model

- Rational Unified Process (RUP) iterative and incremental approach to develop OO software systems



Requirements
Design
Implementation & Test & Integration & More Design
Final Integration & System Test
Time
Requirements
Design
Implementation & Test & Integration & More Design
Final Integration & System Test
3 weeks (for example)

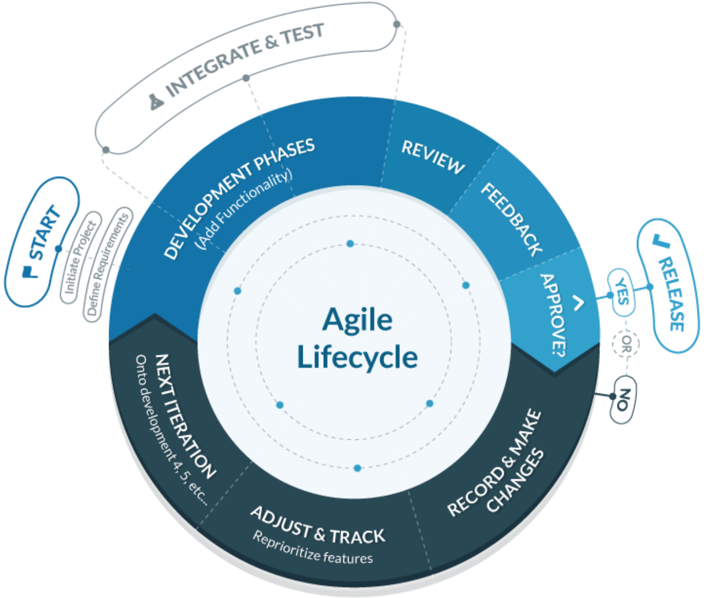
Feedback from iteration N leads to refinement and adaptation of the requirements and design in iteration N+1.

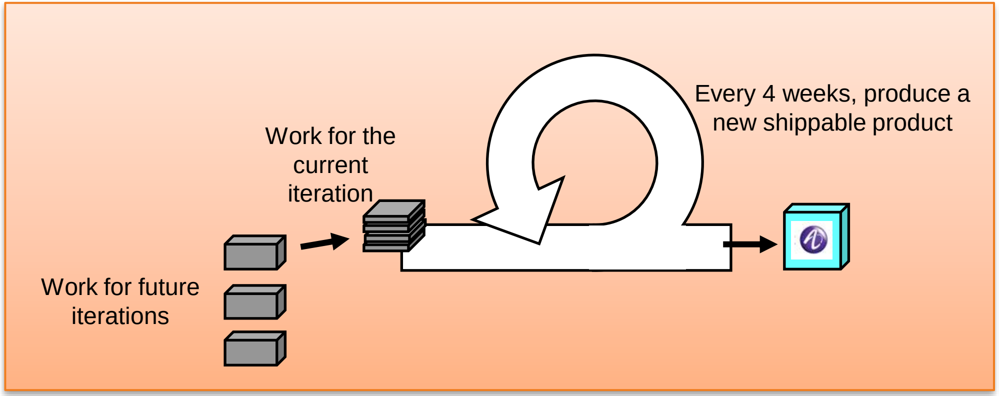
Iterations are fixed in length, or timeboxed.

The system grows incrementally.

Agile Development – Increments

- Software is incrementally developed





Work for future iterations
Work for the current iteration
Every 4 weeks, produce a new shippable product

https://blog.capterra.com/agile-vs-waterfall/

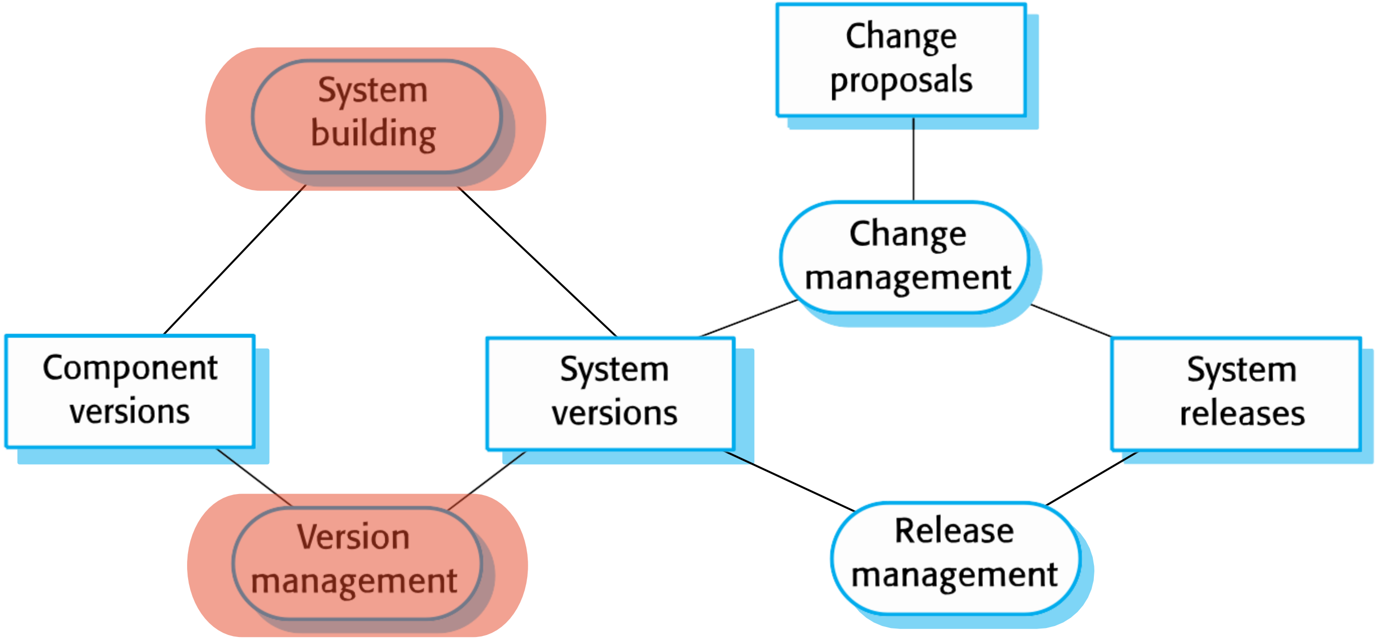
Configuration Management (CM)

- Configuration management (CM) is concerned with the 1 policies, 2 processes 3 and tools for managing changing software systems

- Track of what changes and component versions incorporated into each system version

- Essential for team projects to control changes made by different developers

Configuration Management Activities



Configuration Management Activities

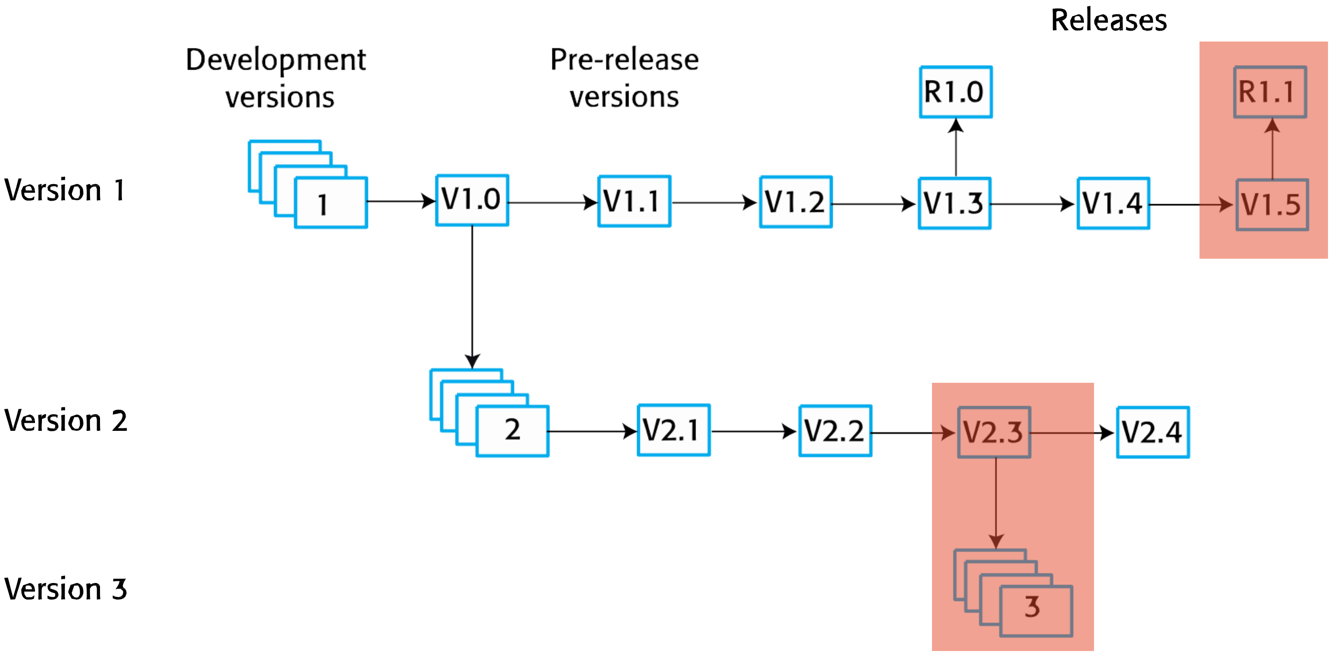
- **System building:** assembling program components, data and libraries, then compiling these to create an executable system

- **Version management:** keeping track of the multiple versions of system components and ensuring that changes made to components by different developers do not interfere with each other

- **Change management:** keeping track of requests for changes to the software from customers and developers, working out the costs and impact of changes, and deciding the changes should be implemented

- **Release management:** preparing software for external release and keeping track of the system versions that have been released for customers

Multi-version System Development

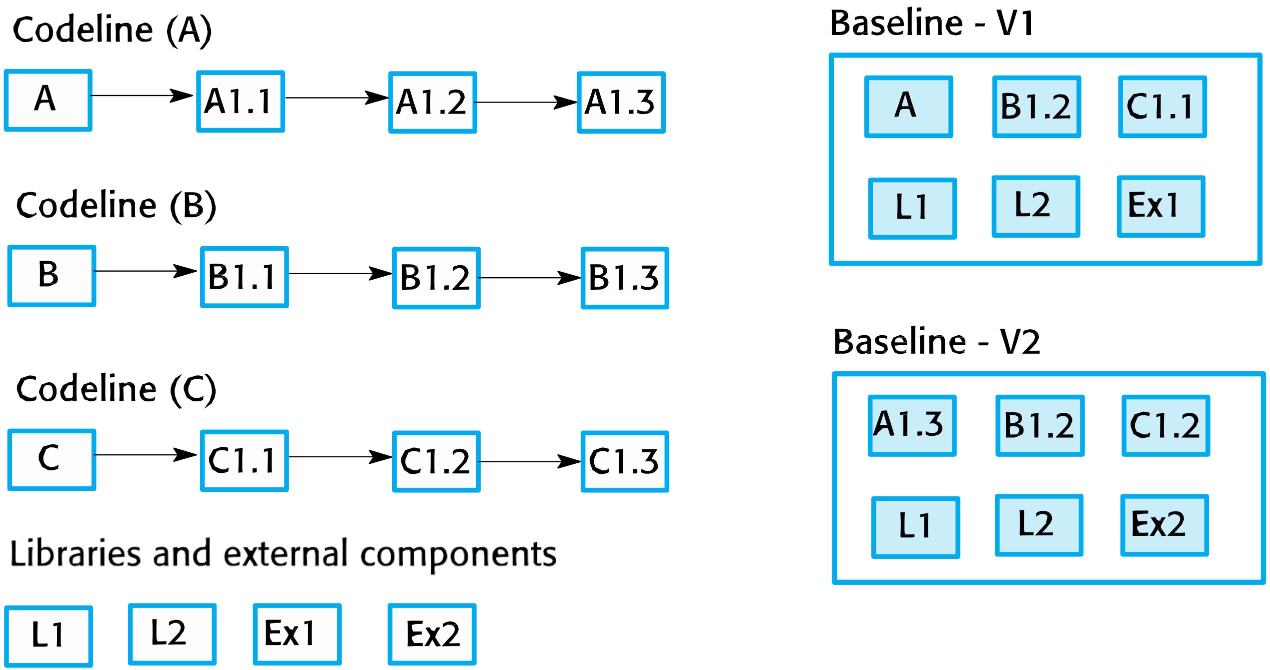


Ian Sommerville. 2016. Software Engineering (10th ed.)

Version Management (VM)

- Keep track of different versions of software components or configuration items and the systems in which these components are used
- Ensuring changes made by different developers to these versions do not interfere with each other
- The process of managing code-lines and baselines

Baselines



Ian Sommerville. 2016. Software Engineering (10th ed.)

Version Management – Codelines and Baselines

- Version Management can be thought of as the process of managing *codelines* and *baselines*

- **Codeline:** a sequence of versions of source code with later versions in the sequence derived from earlier versions
  - System's components often have different versions

- **Baseline:** a definition of a specific system
  - Specifies the component versions that are included in the system plus a specification of the libraries used, configuration files, etc.

Baselines

- Baselines may be specified using a configuration language, which allows you to define what components are included in a version of a particular system.

- Baselines are important because you often have to recreate a specific version of a complete system
  - E.g., individual system versions for different customers. If a customer reports bugs in their system one can recreate the version delivered to a specific customer

Semantic Versioning (SemVer)

- Set of rules and requirements that determine how version numbers should be assigned and incremented for software being developed
  - Semantic numbers; numbers with meaning in relation to a certain version

- Why SemVer?
  - Helps managing versioning numbers in a meaningful and standard way
  - Managing dependencies: the bigger your system grows, the more packages/libraries/plugins you integrate into your software

- Given a version number **MAJOR**.**MINOR**.**PATCH**, increment the:
  1. MAJOR version when you make incompatible API changes,
  2. MINOR version when you add functionality in a backwards-compatible manner,
  3. PATCH version when you make backwards-compatible bug fixes.

https://semver.org/

Semantic Versioning – Example

Stage	Code	Rule	Example
New product	1st release	Start with 1.0.0	1.0.0
Patch Release	Bug fixes, other minor changes	Increment the 3rd digit	1.0.1
Minor Release	New features that do not break existing features	Increment the 2nd digit	1.1.0
Major Release	Changes that break backward compatibility	Increment the 1st digit	2.0.0

https://docs.npmjs.com/getting-started/semantic-versioning

More details on semantic versioning - https://semver.org/

SemVer Example – Dependencies

- “***Firetruck***” library requires a semantically versioned package “***Ladder***”

- ***Firetruck*** uses ***Ladder*** 3.1.0.

- Assume that Ladder3.1.9 is released, can you safely use it safely to release it in your system? Y
- Assume that Ladder3.7.7 is released, can you safely use it safely to release it in your system? Y
- Assume that Ladder4.1.0 is released, can you safely use it safely to release it in your system? N

https://semver.org/#semantic-versioning-specification-semver

Agile Development in CM

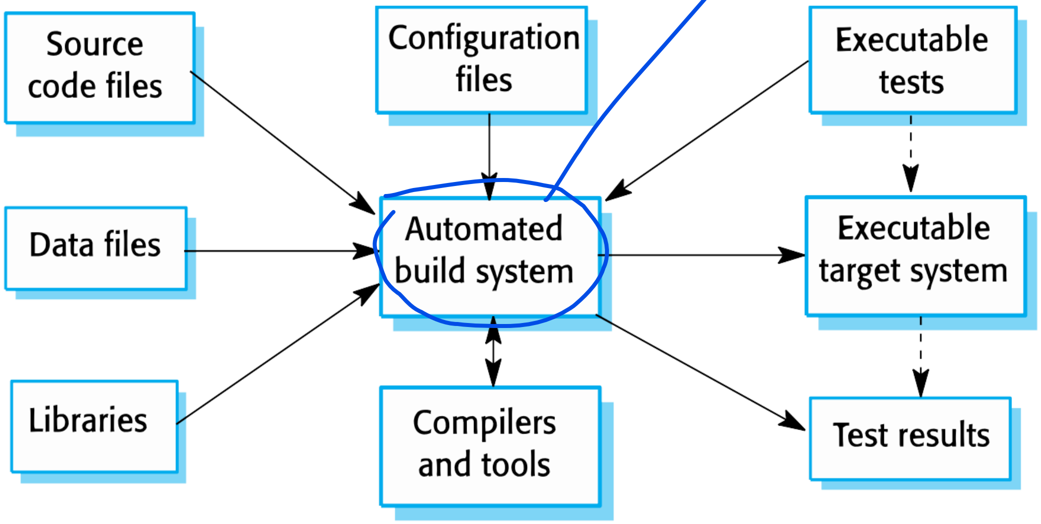
- Agile development, where components and systems are changed several times per day, is impossible without using CM tools

- The definitive versions of components are held in a shared project repository and developers copy these into their own workspace

- They make changes to the code then use system building tools to create a new system on their own computer for testing. Once they are happy with the changes made, they return the modified components to the project repository.

System Building

like gradle



System Building

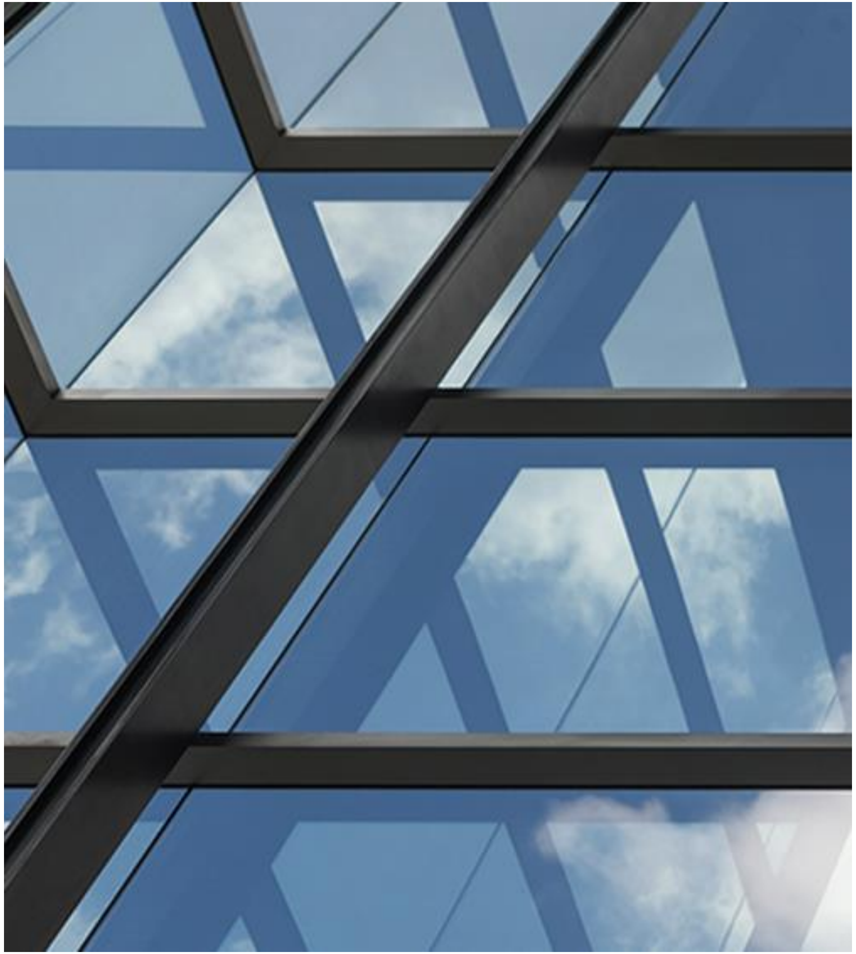
- System building is the process of creating a complete, executable system by compiling and linking the system components, external libraries, configuration files, etc.

- System building tools and version management tools must communicate as the build process involves checking out component versions from the repo managed by the version management system.

- The configuration description used to identify a baseline is also used by the system building tool

Software Build Automation Tools

Ant, Maven, Gradle



Tools for Agile Development

Version Control

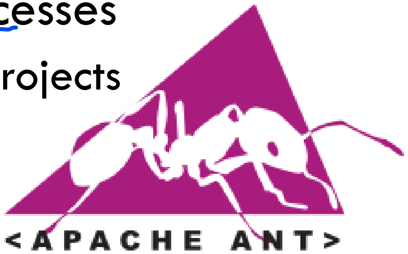
Build Automation

git

gradle

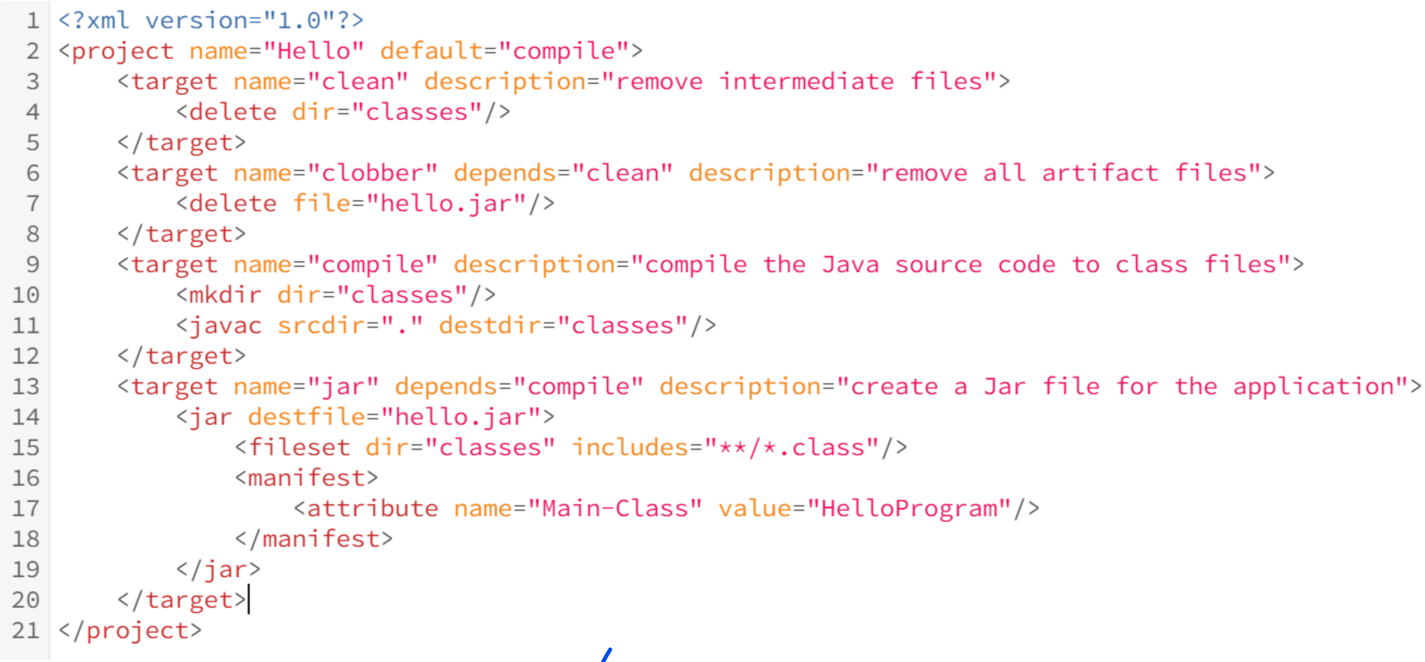
Build Tools – Apache Ant

- Java-based software build tool for automating build processes
  - Requires Java platform and best suited for building Java projects



- Does not impose coding conventions
- Does not impose any heavyweight dependency management framework
- XML to describe the code build process and its dependencies
  - Default *build.xml*

Apache ANT – Example



```xml
<?xml version="1.0"?>
<project name="Hello" default="compile">
    <target name="clean" description="remove intermediate files">
        <delete dir="classes"/>
    </target>
    <target name="clobber" depends="clean" description="remove all artifact files">
        <delete file="hello.jar"/>
    </target>
    <target name="compile" description="compile the Java source code to class files">
        <mkdir dir="classes"/>
        <javac srcdir="." destdir="classes"/>
    </target>
    <target name="jar" depends="compile" description="create a Jar file for the application">
        <jar destfile="hello.jar">
            <fileset dir="classes" includes="**/*.class"/>
            <manifest>
                <attribute name="Main-Class" value="HelloProgram"/>
            </manifest>
        </jar>
    </target>
</project>
```

XML

https://en.wikipedia.org/wiki/Apache_Ant

Apache ANT – Drawbacks

- Too flexible
- Complexity (XML-based build files)
  - Need to specify a lot of things to make simple builds
- No standard structure/layout
  - Developers can create their own structure/layout of the project

Apache Maven

Maven™

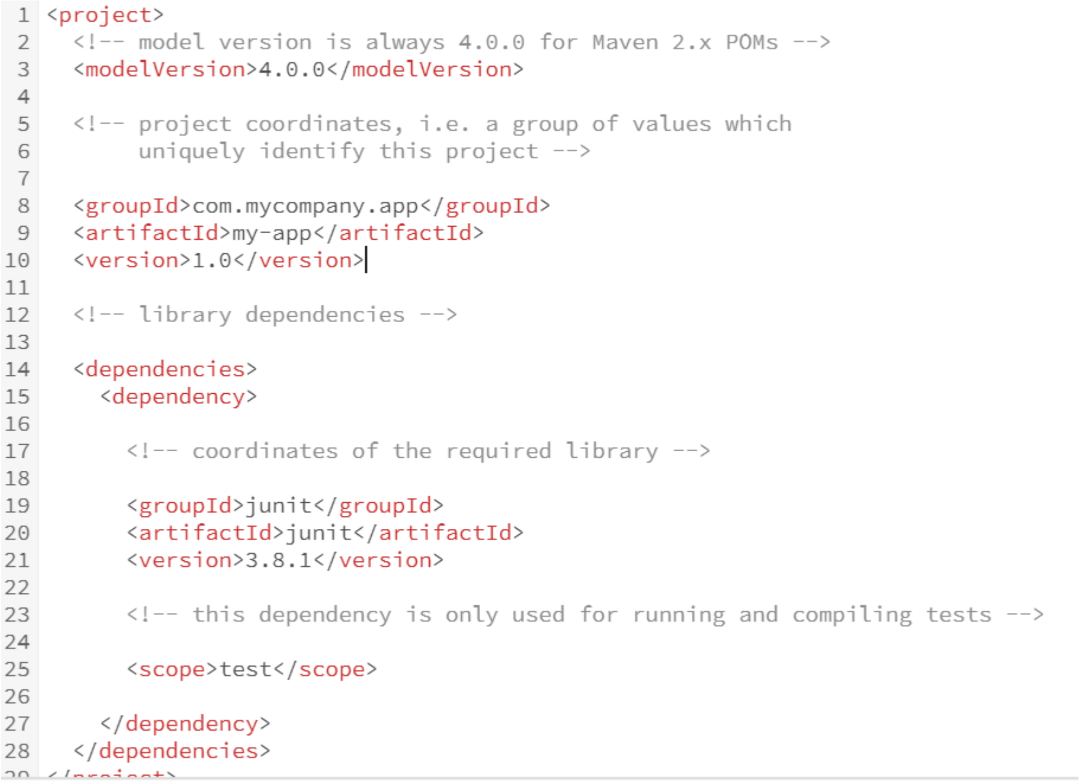
- A build automation tool primarily for java projects
- XML-based description of the software being built
  - Dependencies on other external modules and components, the build order, directories, and required plug-ins
- Central repository (e.g., Maven 2)
- Coding by convention: it uses conventions over configuration for the build procedure
- Supported by Eclipse, IntelliJ, JBuilder, NetBeans IDEs
- Plugin-based architecture
  - Plugin for the .NET framework and native plugins for C/C++ are maintained

Apache Maven – Minimal Example

- Maven projects are configured using *Project Object Model (POM)* stored in a *pom.xml* file

XML

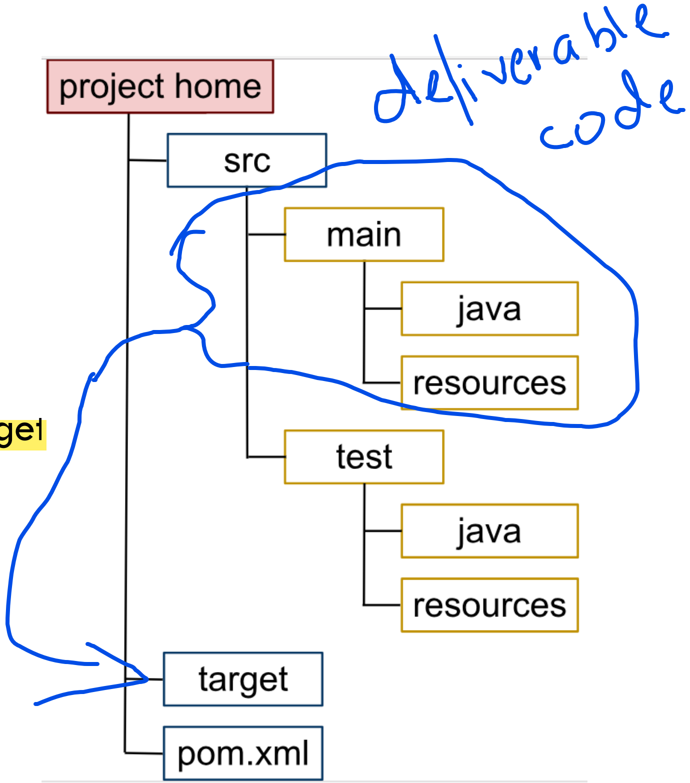
POM



```
<project>
  <!-- model version is always 4.0.0 for Maven 2.x POMs -->
  <modelVersion>4.0.0</modelVersion>

  <!-- project coordinates, i.e. a group of values which
       uniquely identify this project -->

  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1.0</version>

  <!-- library dependencies -->

  <dependencies>
    <dependency>

      <!-- coordinates of the required library -->

      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>

      <!-- this dependency is only used for running and compiling tests -->

      <scope>test</scope>

    </dependency>
  </dependencies>
</project>
```

Apache Maven – Project Structure

- The command *mvn package* will
  - compile all the Java files
  - run any tests
  - package the deliverable code and resources into target (e.g., target/my-app-1.0.jar)



The Maven software tool auto-generated this directory structure for a Java project

https://commons.wikimedia.org/wiki/File:Maven_CoC.svg#/media/File:Maven_CoC.svg

Apache Maven – Central Repository

- Maven uses default Central Repository that maintains required software artefacts (libraries, plug-ins) to manage dependencies
- E.g., project that is dependent on the Hibernate library needs to specify that in the pom.xml project file
  - Maven checks if the referenced dependency is already in the user's local repository
  - It references the dependency from the local repository or
  - Dynamically download the dependency and the dependencies that Hibernate itself needs (transitive dependency) and store them in the user's local repository
- You can configure repositories other than the default (e.g., company-private repository)

Apache Maven – Drawbacks

- Again, XML-based files increase complexity (verbose)

- Rigid; developers are required to understand follow the conventions

Gradle

- Build automation tool that builds upon the concepts of Ant and Maven
  - build conventions, and redefine conventions
  - Project described using Groovy-based Domain Specific Language (DSL)
  - Tasks orders determined using a directed acyclic graph (DAC)
  - Multi-project builds
  - Incremental builds; up to date parts of the build tree do not need to be re-executed
  - Dependency handling (transitive dependency management)

https://en.wikipedia.org/wiki/Gradle

Gradle – Groovy

- Gradle build files are Groovy scripts

- Groovy is a dynamic language of the JVM
  - Can be added as a plug-in
  - Allows developers to write general programming tasks in the build files
  - Relief developers from the lacking control flow in Ant or being forced into plug-in development in Maven to declare nonstandard tasks

relieve*

https://en.wikipedia.org/wiki/Gradle

Gradle – DSL

- Gradle also presents a DSL tailored to the task of building code
  - Not general-purpose or programming language
  - Gradle DSL contains the language needed to describe how to build Java code and create a WAR file from the output
- Gradle DSL is extensible through plug-ins

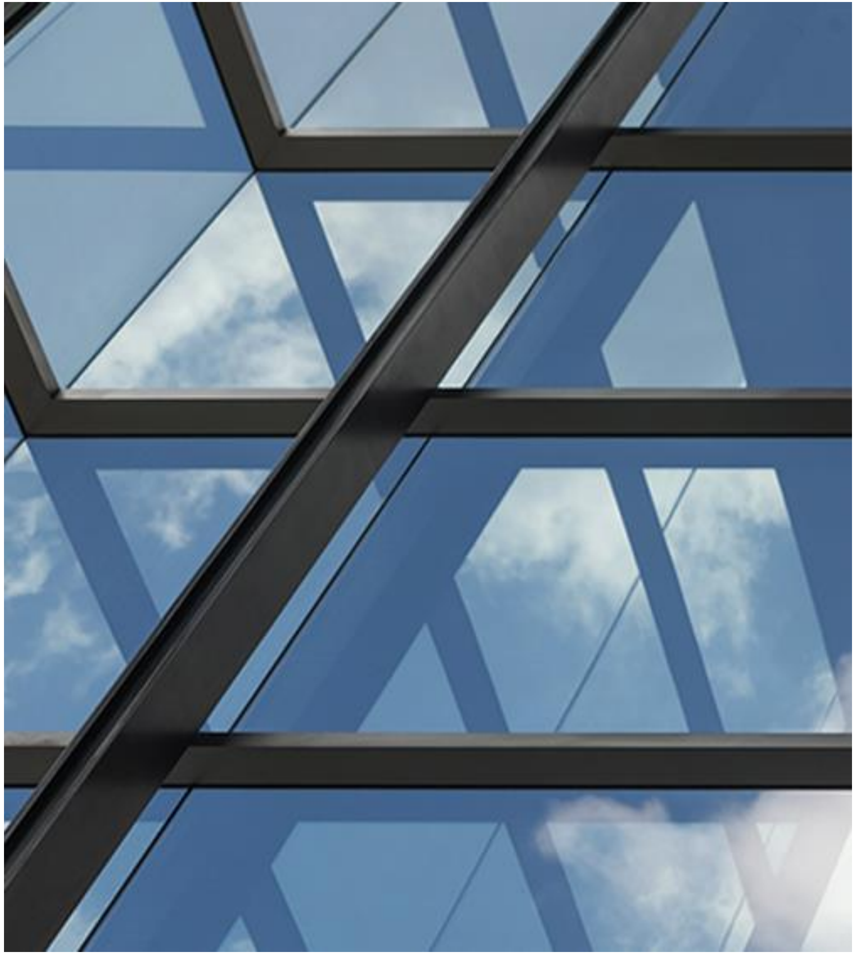
https://en.wikipedia.org/wiki/Gradle

Gradle – Extensible DSL

- Extensibility using plug-ins (if it doesn't have the language to describe what you want your build to do)
  - E.g., describe how to run database migration scripts or deploy code to a set of cloud-based QA servers

- Gradle plug-ins allow:
  - Adding new task definitions
  - Change the behavior of existing tasks
  - Add new objects
  - Create keywords to describe tasks that depart from the standard Gradle categories

Gradle Basics



THE UNIVERSITY OF SYDNEY

Gradle – Tasks

- **Task:** a single atomic piece of work for a build
  - e.g., compiling classes
- **Project:** a composition of several tasks
  - e.g., Creation of a jar file
- Each task has a **name**, which can be used to refer to the task within its owning project, and a **fully qualified path**, which is unique across all tasks in all projects

https://en.wikipedia.org/wiki/Grale

Gradle – Task Actions

- A task is made up of sequence of **Action objects**
  - *Action.execute(T) to execute a task*

- Add actions to a task
  - *Task.doFirst()* or *Task.doLast()*

.doFirst, .doLast are actions of Task.

- Task action exceptions
  - ***StopActionException*** *to* abort execution of the action
  - ***StopExecutionException*** *to* abort execution of the task and continue to the next task

https://en.wikipedia.org/wiki/Grale

Gradle – Simplest Build File Example

Build.gradle

```
task helloWorld << {
        println 'hello, world'
}
```

Results of Hello World build file

```
$ gradle -q helloWorld
hello, world
```

task
call

task
name

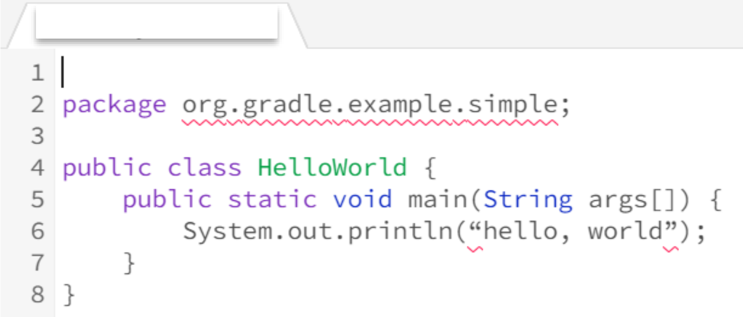
Build.gradle

```
task hello << {
print 'hello, '
}
task world(dependsOn: hello) << {
println 'world'
}
```

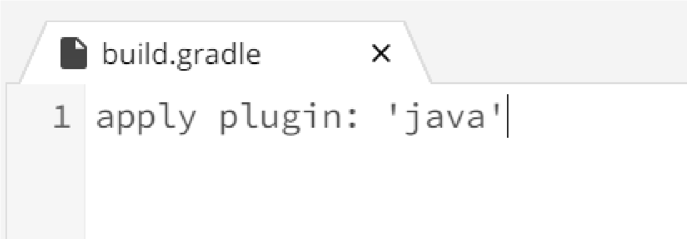
execute the second task, **world**

```
$ gradle -q world
hello, world
```

Gradle – Simplest Build File for Java Example (1)



```

package org.gradle.example.simple;

public class HelloWorld {
    public static void main(String args[]) {
        System.out.println(“hello, world”);
    }
}
```



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

Simplest possible Gradle file for java

```
.
├── build.gradle
    └── src
        └── main
            └── java
                └── org
                    └── gradle
                        └── example
                            └── simple
                                └── HelloWorld.java
```

Project layout of HelloWorld.java

Gradle – Simplest Build File for Java Example (2)

gradle build

Gradle automatically introduces number of tasks for us to run

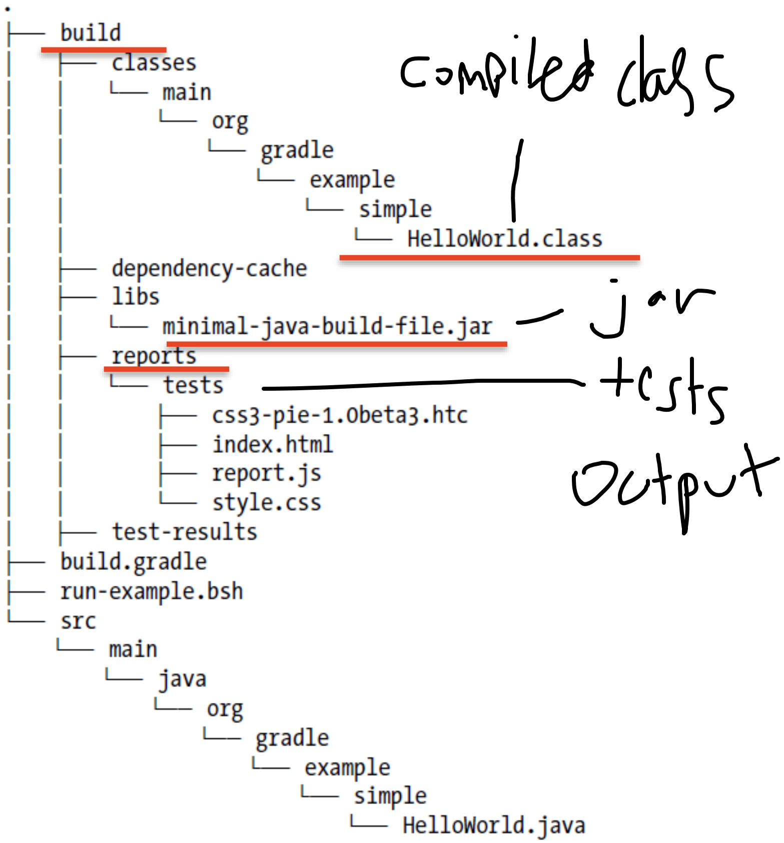
Note:

- Class files generated and place in a directory
- Test report files (for unit test results)
- JAR built using the project directory

Run HelloWorld Java

$ java -cp build/classes/main/ org.gradle.example.simple.HelloWorld

hello, world



```
.
├── build
│   ├── classes
│   │   └── main
│   │       └── org
│   │           └── gradle
│   │               └── example
│   │                   └── simple
│   │                       └── HelloWorld.class
│   ├── dependency-cache
│   ├── libs
│   │   └── minimal-java-build-file.jar
│   ├── reports
│   │   └── tests
│   │       ├── css3-pie-1.0beta3.htc
│   │       ├── index.html
│   │       ├── report.js
│   │       └── style.css
│   └── test-results
├── build.gradle
├── run-example.bsh
└── src
    └── main
        └── java
            └── org
                └── gradle
                    └── example
                        └── simple
                            └── HelloWorld.java
```

Project Layout of Hello World Java After Build

Gradle – Build Lifecycle

- Phases of executing a build file in gradle:

1 - **Initialization:** projects are to participate in the build

2 - **Configuration:** task objects are assembled into an internal object model called Directed Acyclic Graph (DAG)

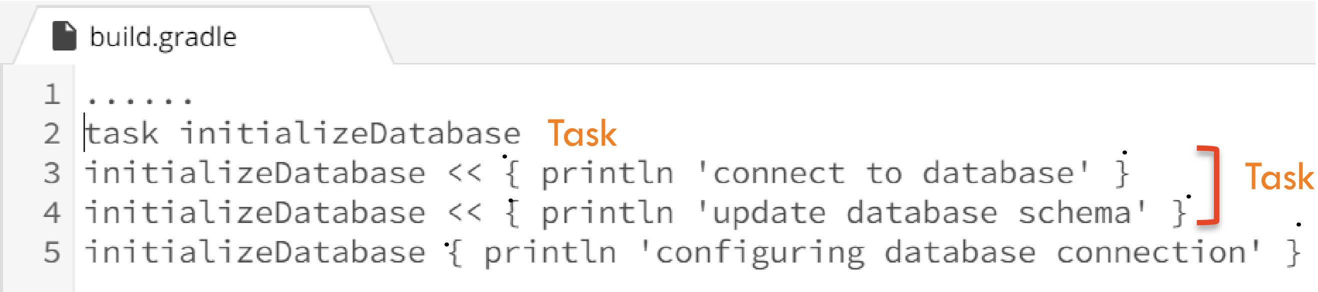
3 - **Execution:** build tasks are executed in the order required by their dependency relationship

Gradle – Task Configuration

- **Configuration block:** to setup variables and data structures needed by the task action when it runs in the build
  - Make tasks rich object model populated with information about the build
  - Runs during gradle's configuration lifecycle when task actions executed

- **Closure:** a block of code specified by curly braces
  - Holding blocks of configuration and build actions

Gradle – Task Configuration Example



build.gradle

```
......
task initializeDatabase
initializeDatabase << { println 'connect to database' }
initializeDatabase << { println 'update database schema' }
initializeDatabase { println 'configuring database connection' }
```

Task

Task actions

Configuration block

Closure

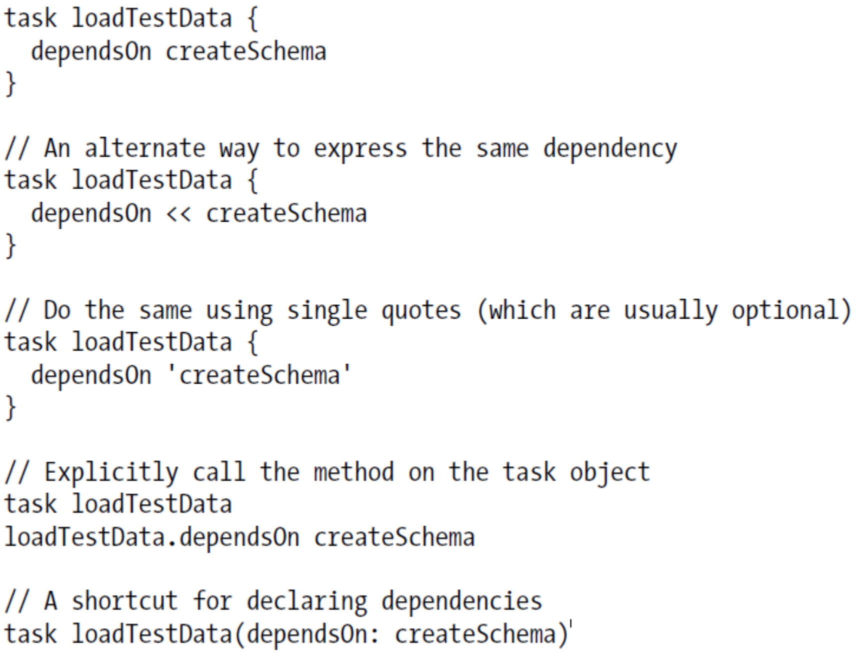
Gradle – Tasks are Objects

- Every task is represented internally as an **object**
  - Task's methods and properties
  - Gradle creates an internal object model of the build before executing it
  - Each new task is of *DefaultTask* type - task type can be changed
  - *DefaultTask* contains functionality required for them to interface with Gradle project model

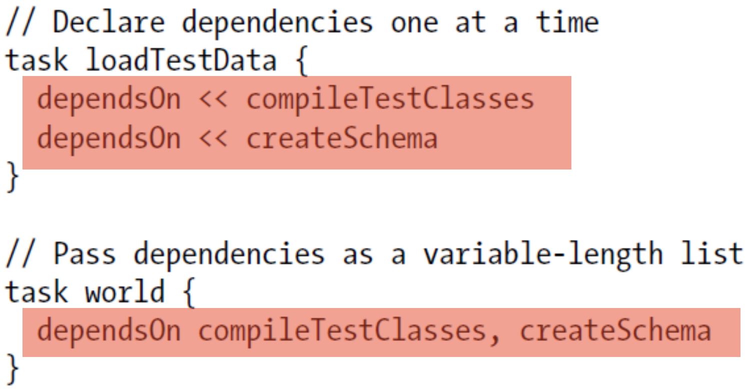
Gradle – Methods of Default Task

Method	Description
dependsOn(task)	Adds a task as a dependency of the calling task. A depended-on task will always run before the task that depends on it
doFirst(closure)	Adds a block of executable code to the beginning of a task's action. During the execution phase, the action block of every relevant task is executed.
doLast(closure)	Appends behavior to the end of an action
onlyIf(closure)	Expresses a predicate which determines whether a task should be executed. The value of the predicate is the value returned by the closure.

Gradle – dependsOn() Example



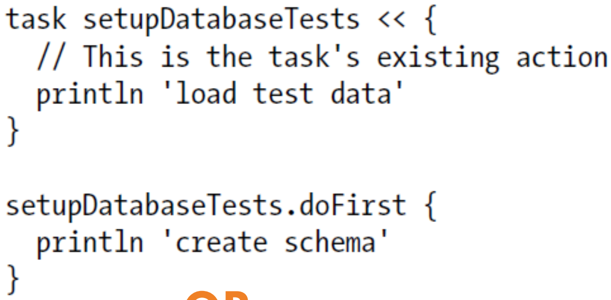
```
task loadTestData {
  dependsOn createSchema
}

// An alternate way to express the same dependency
task loadTestData {
  dependsOn << createSchema
}

// Do the same using single quotes (which are usually optional)
task loadTestData {
  dependsOn 'createSchema'
}

// Explicitly call the method on the task object
task loadTestData
loadTestData.dependsOn createSchema

// A shortcut for declaring dependencies
task loadTestData(dependsOn: createSchema)'
```



```
// Declare dependencies one at a time
task loadTestData {
  dependsOn << compileTestClasses
  dependsOn << createSchema
}

// Pass dependencies as a variable-length list
task world {
  dependsOn compileTestClasses, createSchema
}
```

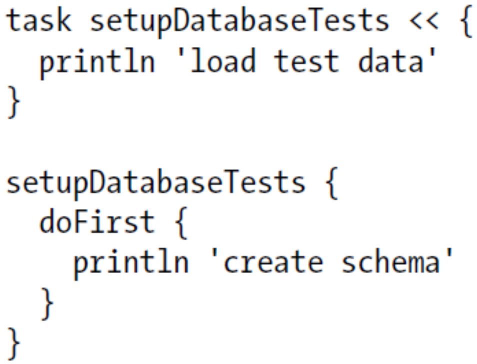
Different ways to call the *dependsOn* method

Gradle – doFirst() Example



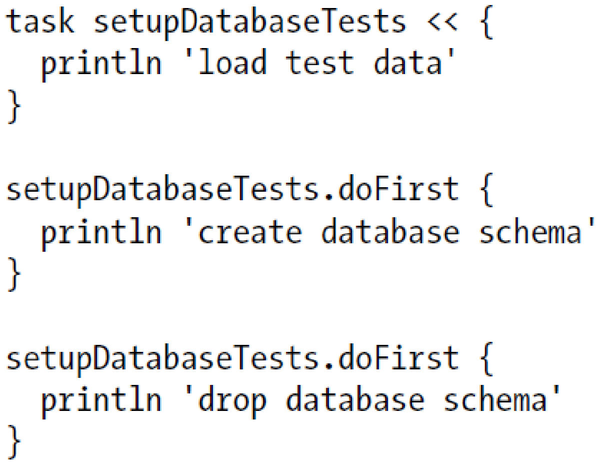
```
task setupDatabaseTests << {
  // This is the task's existing action
  println 'load test data'
}

setupDatabaseTests.doFirst {
  println 'create schema'
}
```

OR



```
task setupDatabaseTests << {
  println 'load test data'
}

setupDatabaseTests {
  doFirst {
    println 'create schema'
  }
}
```

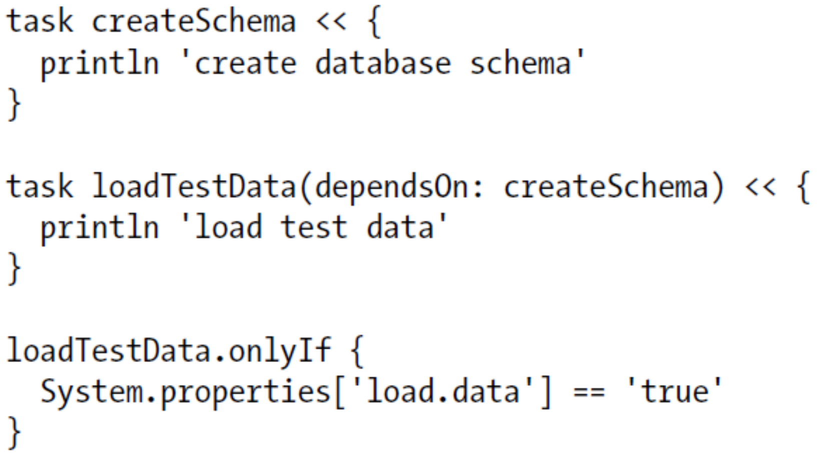
Call the doFirst on the task object (top) and inside task's configuration block (bottom)



```
task setupDatabaseTests << {
  println 'load test data'
}

setupDatabaseTests.doFirst {
  println 'create database schema'
}

setupDatabaseTests.doFirst {
  println 'drop database schema'
}
```

Repeated calls to the doFirst method are additive.

Gradle – onlyIf() Example



```
task createSchema << {
  println 'create database schema'
}

task loadTestData(dependsOn: createSchema) << {
  println 'load test data'
}

loadTestData.onlyIf {
  System.properties['load.data'] == 'true'
}
```

Using onlyIf method to do simple system property tests

- *onlyIf* method can be used to switch individual tasks on and off using any logic you can express in Groovy code

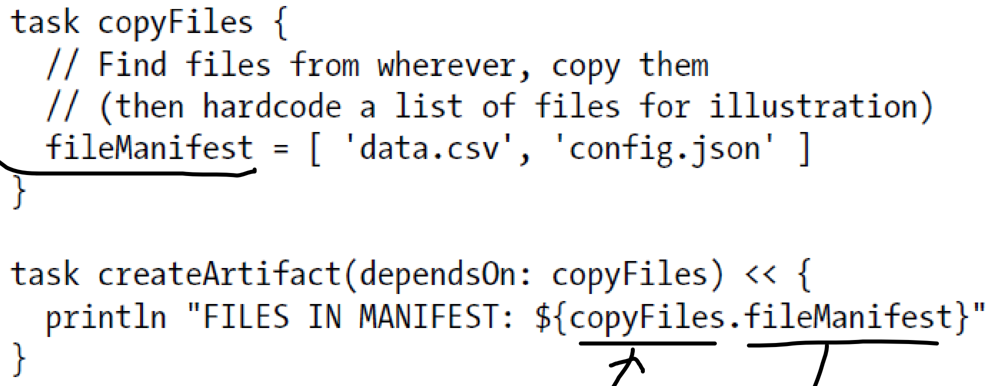
- E.g., read files, call web services, check security credentials

Gradle – Default Task's Properties

Method	Description
didWork	A Boolean property indicating whether the task completed successfully
enabled	A Boolean property indicating whether the task will execute.
path	A string property containing the fully qualified path of a task (levels; DEBUG, INFO, LIFECYCLE, WARN, QUIET, ERROR)
logger	A reference to the internal Gradle logger object
logging	The logging property gives us access to the log level
temporaryDir	Returns a File object pointing to a temporary directory belonging to this build file. It is generally available to a task needing a temporary place in to store intermediate results of any work, or to stage files for processing inside the task
description	a small piece of human-readable metadata to document the purpose of a task

Gradle – Dynamic Properties

- Properties (other than built-in ones) can be assigned to a task
- A task object functions can contain other arbitrary property names and values we want to assign to it (do not use built-in property names)



```
task copyFiles {
  // Find files from wherever, copy them
  // (then hardcode a list of files for illustration)
  fileManifest = [ 'data.csv', 'config.json' ]
}

task createArtifact(dependsOn: copyFiles) << {
  println "FILES IN MANIFEST: ${copyFiles.fileManifest}"
}
```

property value

task name

Gradle Task Types – Copy

- A copy task copies files from one place into another

to define a copy task!

```
task copyFiles(type: Copy) {
from 'resources'
into 'target'
include '**/*.xml', '**/*.txt',
'**/*.properties'
}
```

Note: the *from*, *into*, and *include* methods are inherited from the Copy

Gradle Task Types – Jar

- A *Jar* task creates a jar file from source files
- The *Java plug-in's jar* creates a task of this type
- It packages the main source set and resources together with a trivial manifest into a jar bearing the project's name in the build/libs directory
- highly customizable

Gradle Task Types – Jar Example

```
apply plugin: 'java'
task customJar(type: Jar) {
manifest {
attributes firstKey: 'firstValue', secondKey: 'secondValue'
}
archiveName = 'hello.jar'
destinationDir = file("${buildDir}/jars")
from sourceSets.main.classes
}
```

Gradle Task Types – JavaExec

- A *JavaExec* task runs a Java class with a main() method

- Tries to take the hassle away and integrate command-line Java invocations into your build

Gradle Task Types – JavaExec Example

```
apply plugin: 'java'
repositories {
mavenCentral()
}
dependencies {
runtime 'commons-codec:commons-codec:1.5'
}
task encode(type: JavaExec, dependsOn: classes) {
main = 'org.gradle.example.commandline.MetaphoneEncoder'
args = "The rain falls mainly in the plain".split().toList()
classpath sourceSets.main.classesDir
classpath configurations.runtime
}
```

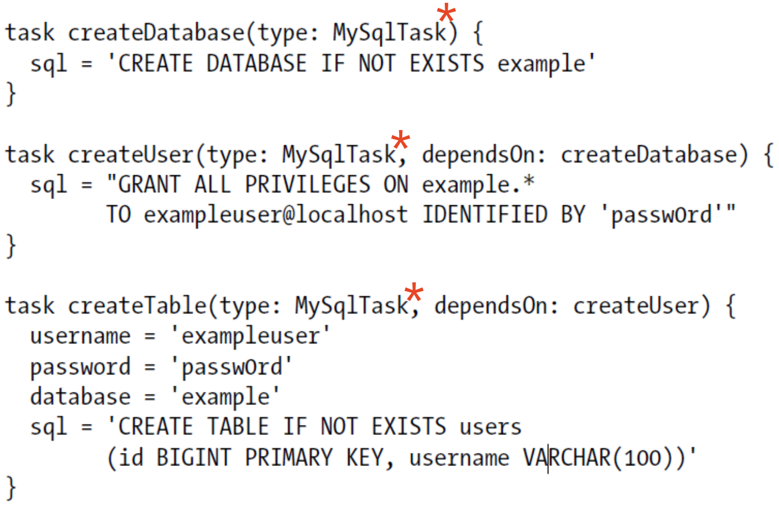
Gradle Custom Task Types

- Gradle's built-in tasks might not be sufficient for all scenarios

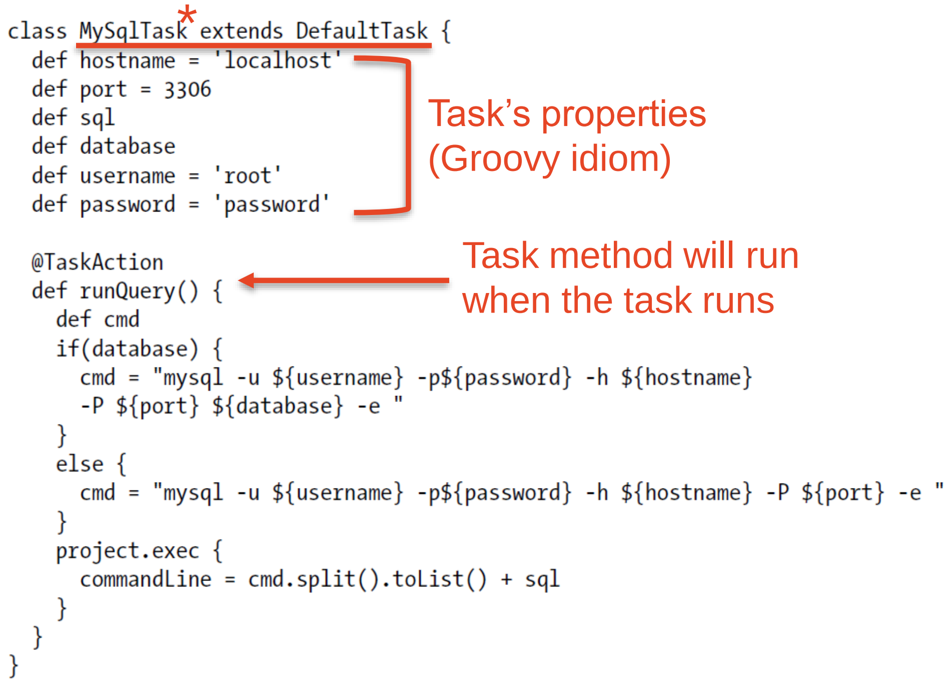
- Gradle allows defining custom task types in:
  - The **Build File** and must extend the *DefaultTask* class or one of its descendants
  - The **Source Tree:** when the task's logic is significant has its own class hierarchy and might rely on external interface
    - When the custom task logic outgrows the build file, it can be migrated to the **buildSrc** directory at the project root
    - This directory is automatically compiled and added to the build classpath

Gradle Custom Task Types – Build File

– Suppose your build file needs to issue arbitrary queries against a MySQL database



```
task createDatabase(type: MySqlTask) {
  sql = 'CREATE DATABASE IF NOT EXISTS example'
}

task createUser(type: MySqlTask, dependsOn: createDatabase) {
  sql = "GRANT ALL PRIVILEGES ON example.*
        TO exampleuser@localhost IDENTIFIED BY 'passw0rd'"
}

task createTable(type: MySqlTask, dependsOn: createUser) {
  username = 'exampleuser'
  password = 'passw0rd'
  database = 'example'
  sql = 'CREATE TABLE IF NOT EXISTS users
        (id BIGINT PRIMARY KEY, username VARCHAR(100))'
}
```



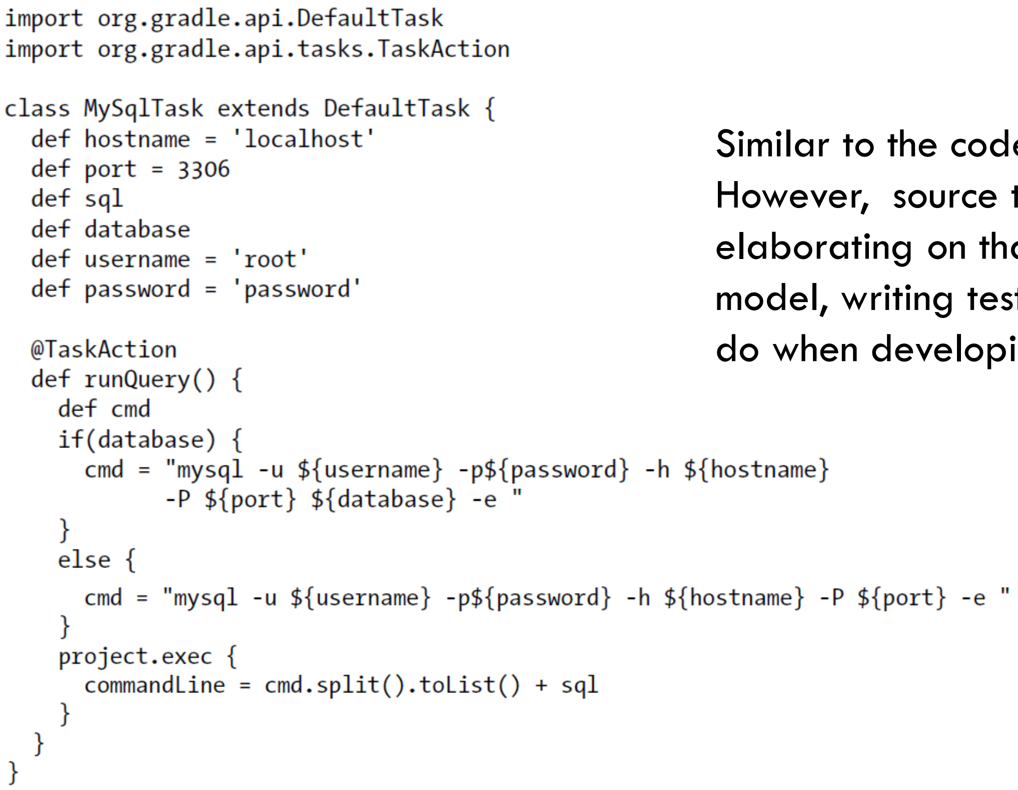
```
class MySqlTask extends DefaultTask {
  def hostname = 'localhost'
  def port = 3306
  def sql
  def database
  def username = 'root'
  def password = 'password'

  @TaskAction
  def runQuery() {
    def cmd
    if(database) {
      cmd = "mysql -u ${username} -p${password} -h ${hostname}
      -P ${port} ${database} -e "
    }
    else {
      cmd = "mysql -u ${username} -p${password} -h ${hostname} -P ${port} -e "
    }
    project.exec {
      commandLine = cmd.split().toList() + sql
    }
  }
}
```

Task's properties (Groovy idiom)

Task method will run when the task runs

* Actual build tasks inherits MySQLType's properties and actions

Gradle Custom Task Types – Source Tree

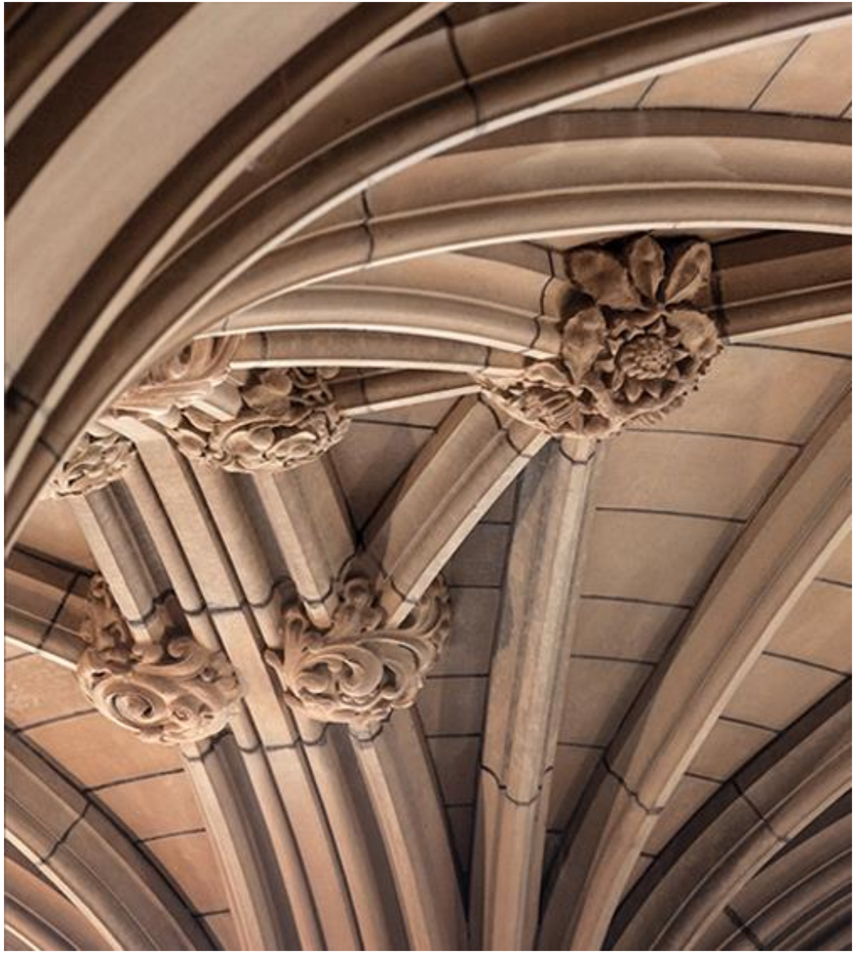


```
import org.gradle.api.DefaultTask
import org.gradle.api.tasks.TaskAction

class MySqlTask extends DefaultTask {
  def hostname = 'localhost'
  def port = 3306
  def sql
  def database
  def username = 'root'
  def password = 'password'

  @TaskAction
  def runQuery() {
    def cmd
    if(database) {
      cmd = "mysql -u ${username} -p${password} -h ${hostname}
            -P ${port} ${database} -e "
    }
    else {
      cmd = "mysql -u ${username} -p${password} -h ${hostname} -P ${port} -e "
    }
    project.exec {
      commandLine = cmd.split().toList() + sql
    }
  }
}
```

Similar to the code in the build script (previous example. However, source tree provides a robust way for elaborating on that simple task behavior, growing an object model, writing tests, and doing everything else we normally do when developing software

Gradle – Plug-ins

Java Plug-in



Gradle – Java Plug-in

- A plug-in is an extension to Gradle which configures projects
- **Java plug-in** adds some tasks to your project which will compile and unit test your Java source code, and bundle into a JAR
  - Convention-based; default values for many aspects of the project are pre-defined
    - In your build.gradle: *apply plugin : 'java'* ←
  - Can customize projects if you do not follow the convention

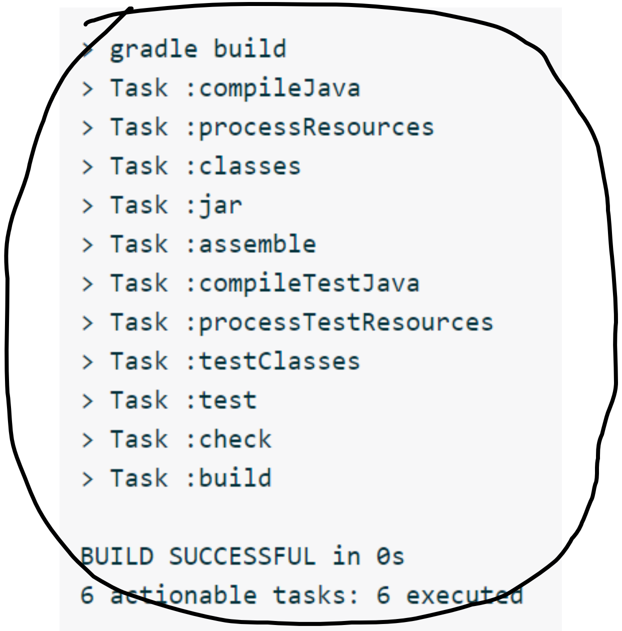
Gradle – Java Plug-in (Project Structure)

- Gradle expects to find production source code under ***src/main/java*** *and test source code under* ***src/test/java***
- Files under ***src/main/resources*** will be included in the JAR as resources
- Files under ***src/test/resources*** will be included in the classpath used to run tests
- All output files are created under the build directory, with the JAR file will end up in the ***build/libs*** directory

https://docs.gradle.org/current/userguide/java_plugin.html

Gradle – Java Plug-in (Project Build)

- *Java plug-in* will add a few tasks
- Run *gradle tasks* to list the tasks of a project
- Gradle build will compile and create a JAR file containing main classes and resources – run *gradle build*



```
> gradle build
> Task :compileJava
> Task :processResources
> Task :classes
> Task :jar
> Task :assemble
> Task :compileTestJava
> Task :processTestResources
> Task :testClasses
> Task :test
> Task :check
> Task :build

BUILD SUCCESSFUL in 0s
6 actionable tasks: 6 executed
```

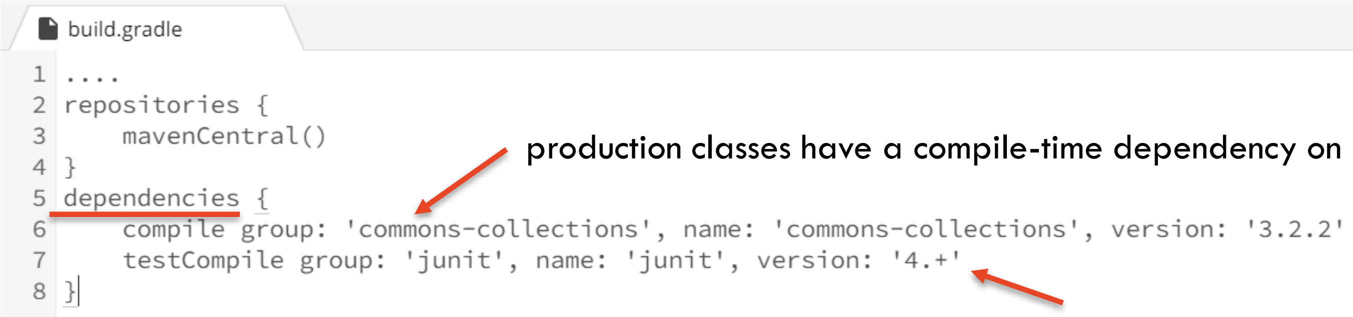
Example of output of gradle build

Gradle – Java Plug-in (Project Build)

- **clean**
  - Deletes the build directory, removing all built files
- **assemble**
  - Compiles and jars your code, but does not run the unit tests
  - Other plugins add more artefacts to this task;
- **check**
  - Compiles and tests your code
  - Other plugins add more checks to this task; e.g., *checkstyle* plugin to run checkstyle against your source code

Gradle Java Plug-in – Dependencies

- Reference external JAR files that the project is dependent on:
  - JAR files in a repository (artefacts/dependencies needed for a project)
  - Different repositories types are supported in Gradle (see Gradle Repository Types)
  - Example (using Central Maven Repository)



```
....
repositories {
    mavenCentral()
}
dependencies {
    compile group: 'commons-collections', name: 'commons-collections', version: '3.2.2'
    testCompile group: 'junit', name: 'junit', version: '4.+'
}
```

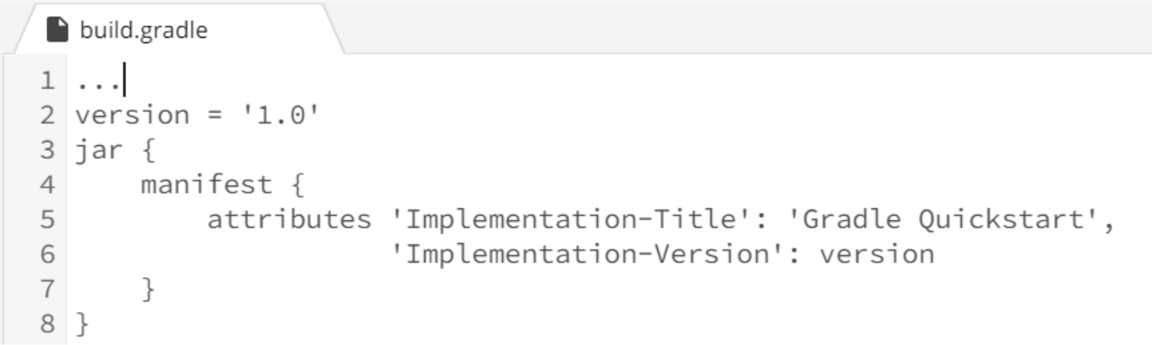
production classes have a compile-time dependency on commons collections

Test classes have a compile-time dependency on junit

https://docs.gradle.org/current/userguide/java_plugin.html

Gradle – Java Plug-in (Project Customization)

- The Java plug-in adds many properties with default values to a project
- Customize default values to suit project needs
- Use *Gradle properties* to list properties of a project



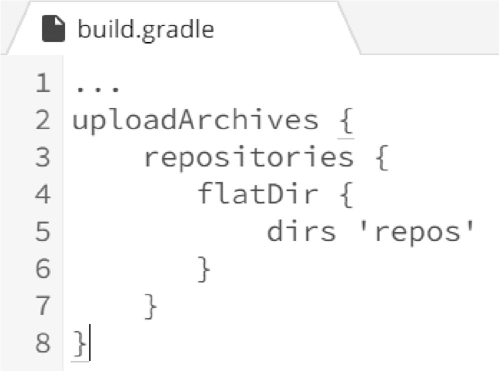
```
...
version = '1.0'
jar {
    manifest {
        attributes 'Implementation-Title': 'Gradle Quickstart',
                   'Implementation-Version': version
    }
}
```

https://docs.gradle.org/current/userguide/java_plugin.html

Gradle – Java Plug-in (Publish JAR file)

- Artefacts such as JAR files can be published to repositories

- To publish a JAR file
  - **gradle *uploadArchives***



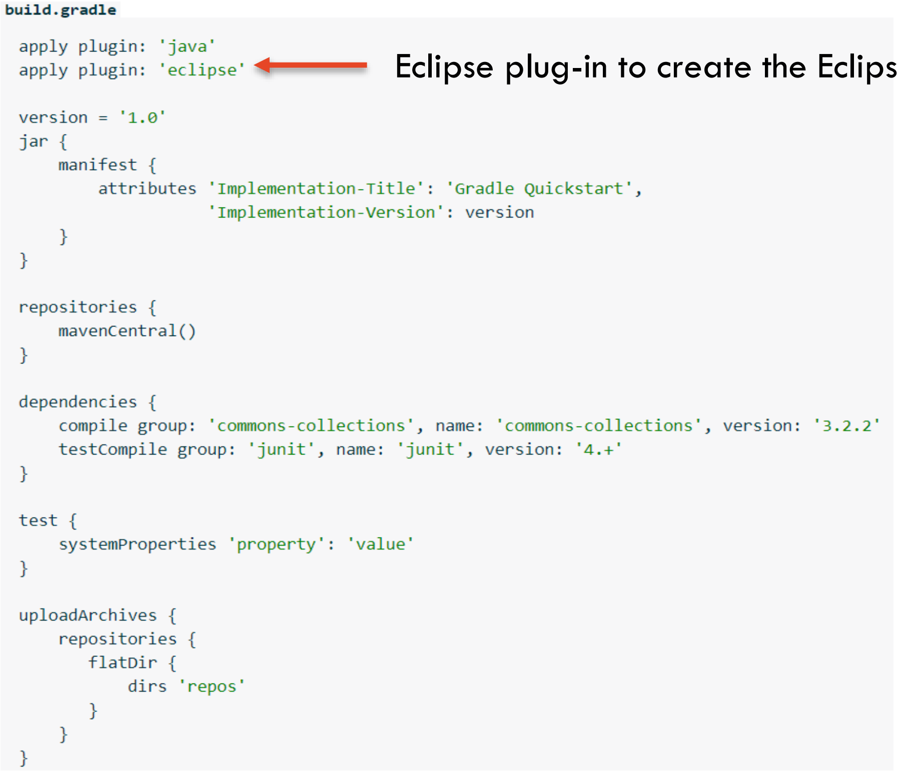
build.gradle

```
...
uploadArchives {
    repositories {
        flatDir {
            dirs 'repos'
        }
    }
}
```

Publish a JAR file to a local repository

https://docs.gradle.org/current/userguide/java_plugin.html

Gradle – Complete Build file for Java



build.gradle

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

version = '1.0'
jar {
    manifest {
        attributes 'Implementation-Title': 'Gradle Quickstart',
                   'Implementation-Version': version
    }
}

repositories {
    mavenCentral()
}

dependencies {
    compile group: 'commons-collections', name: 'commons-collections', version: '3.2.2'
    testCompile group: 'junit', name: 'junit', version: '4.+'
}

test {
    systemProperties 'property': 'value'
}

uploadArchives {
    repositories {
       flatDir {
           dirs 'repos'
       }
    }
}
```

Eclipse plug-in to create the Eclipse-specific descriptor files, like .project

https://docs.gradle.org/current/userguide/java_plugin.html

References

- Ian Sommerville 2016. Software Engineering: Global Edition (3rd edition). Pearson, Englad

- Wikipedia, Apache Ant, https://en.wikipedia.org/wiki/Apache_Ant
- Wikipedia, Apache Maven, https://en.wikipedia.org/wiki/Apache_Maven
- Apache Maven, https://maven.apache.org/
- Gradle documentation, https://docs.gradle.org/current/userguide/
- Tim Berglund and Matthew McCullough. 2011. Building and Testing with Gradle (1st ed.). O'Reilly Media, Inc.