Software Engineering Concepts (COMP3003)

4. Build Engineering

The **build process** involves, essentially, taking stuff that software engineers develop and turning it into artefacts that can be tested and (potentially) released. The output of that process – all the compiled, packaged code, resources, documentation, etc. – is called a **build**.

When you first learn programming, the compiler does all the building that you need. It takes your source code and gives you executable compiled code that you can run and test. End of story, right?

In fact, things get much more complicated than that. The way that some projects need to be built can be so complex that it justifies adopting the term **build engineering**, which describes the job of setting up and maintaining the build process.

Build engineering is a kind of meta-software-development. Setting up a complex build process often involves writing code – not production code, but **build logic**. You may think of this as a small piece of software whose function is to build the main piece of software. So, although your IDE may present you with a fixed set of "build options", or the like, build engineering in general is an open-ended problem.

What makes the build process complicated? Here are some reasons in brief (which we'll expand on):

- Use cases – build logic can have lots of them. You might want to perform a complete build from scratch. You might want to perform an "incremental" build that just updates the last build. You might want to execute the unit tests, or package up the software, or generate the documentation, or simply verify that external dependencies are met. Each of these requires some combination of the other steps to be completed first.
- Pre-processing and code generation. If you need to do such things, they represent additional tasks to perform prior to actually compiling the code.
- Resource packaging[a]. Your application may have a range of internal resources (e.g. icons, HTML files, etc.) that need to be packaged up in some way that allows the code to access them easily.
- External dependencies. Applications often depend on libraries and other software packages, and these may in turn depend on more libraries. Moreover, these dependencies may be specific to certain parts of the build process; i.e., you need certain libraries to *compile* your application, but others to *test* it, and different (or additional) ones again to *package* it up. Dependencies can also differ from platform to platform.
- Sub-projects. Your project may comprise many different parts, each of which has its own special requirements regarding code generation, resources, and dependencies. Breaking down a project into parts can make it more manageable, but each of these parts will need to be handled by the build logic.

[a] In our discussion of multithreading, we talked about shared resources. The kinds of resources we deal with at build time *can* be shared, but they are also almost certainly immutable.

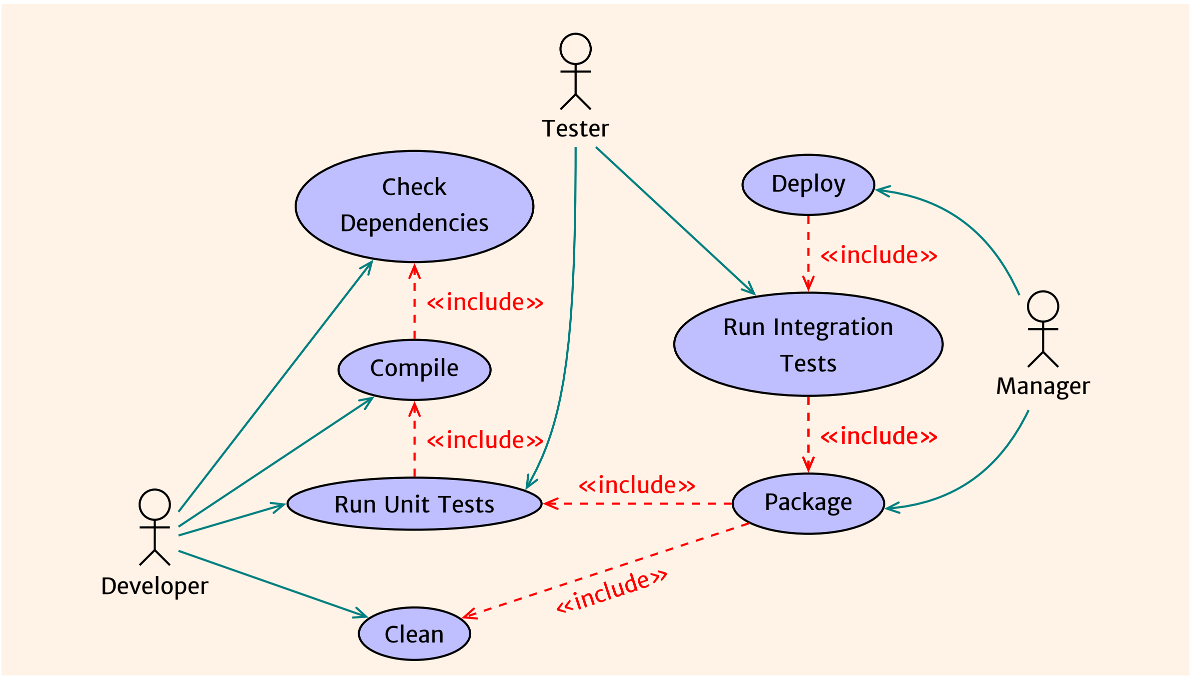

Figure 1: A simplified illustration of build users and tasks (as a UML use case diagram). Real build systems can be much more complex!

- Portability. You generally need to build a different version of your software for each platform on which it will run, all from the same code base. (JVM-based languages can simplify this, but developing cross-platform software is a significant challenge in languages like C and C++.)
- Parallel versions. Even for a single platform, you may still need to produce several different simultaneous versions of your software, each having different sets of features or other characteristics. For instance, you might have a "free" and a "paid" version.
- Generalisation. Last but not least, if your build logic is already a bit complex (as a result of the above), you may want to think about its maintainability. How can you minimise any changes required to the build logic when the production code changes? To do that, you could have your build logic treat the production code as a kind of input, rather than as a hard-coded list of files. That is, your build logic doesn't just perform the required tasks, but to some extent it needs to *find out* what needs to be done in the first place. You might also think of this as decoupling the build logic from the production code.

1 Build Users and Use Cases

A **build user** is someone who uses the build logic (but who does not necessarily modify it). Build users include the actual developers of the software, but also, for instance, testers and system administrators. For open-source projects, anyone "downstream", who might be using your software as a starting point for their own customised version could also be a build user. A user of the main application is *generally not* a build user (unless you expect your users to be extremely technically knowledgeable).

Build tools all recognise a range of different "tasks" (or "phases"), and these are essentially the build logic's use cases. Figure 1 illustrates how build users and tasks may interact.

To refresh your memory, the «include» relationship indicates where one use case (task, in this case) causes the execution of another. For instance, in order to run unit tests, we must first compile the code. You can see, therefore, that there is a chain of tasks that begins with checking dependencies and ends with deployment. To execute the Deploy task, you must first complete all the others (according to the diagram).

Table 1: A few commonly used build tools.

Build tool	Typical environment	Build logic language
Autotools	GNU/Linux, predominantly C-based software	M4 (macro language)
CMake	Open source C/C++ software	CMake scripting language
Ant/Ivy	Java and JVM languages	XML-based, plus various JVM-based languages (BeanShell, Groovy, JRuby, Jython, etc.)
Maven	Java and JVM languages	XML-based, plus Java
Gradle	Java and JVM languages, and particularly Android projects	Groovy / Kotlin
MSBuild	Windows/.NET	XML-based
NAnt	Windows/.NET	XML-based
xcodebuild	OS X and iOS	No specific language

However, notice that each task is also connected to a type of build user. A developer can choose to compile *without* running the unit tests (those «include» relationships being one way). The diagram shows build users to only using a specific set of tasks (depending on their role), but really there's no need for the build logic to actually restrict build users like this.

Also, and very importantly, **it's not always like this!** Well, not completely, anyway. Figure 1 shows only *one possible* arrangement of build logic. Different situations may involve different kinds of build users, and will call for different tasks and different relationships between them. (Who's to say that deployment depends on running integration tests, for instance?) However, hopefully you get the basic idea.

2 General Tool Support

You do not strictly need a tool (other than a compiler or interpreter) in order to write build logic. Ultimately, it is simply a program in itself.

However, there are many recurring themes in build logic, and there are many tools that take care of the more conventional tasks for you. Table 1 lists some of them. Each one is often focused on a particular section of the software engineering community, although most of them *could* be used in almost any situation.

Each build tool has its own preferred language for writing build logic, but ultimately you can write build logic in any general-purpose programming/scripting language.

Tools like these exist because large parts of the build process are very similar across different projects, and having a range of highly similar situations is the basis for automation. However, different build tools can have quite different approaches.

A build tool works by first determining everything that *could* happen – all the build-related tasks relevant to the project and their dependencies. It then figures out which of these tasks actually need to be run, and in what order, and carries them out. Gradle (perhaps more than other build tools) makes this distinction explicit in its "configuration" and "execution" phases.

To make this work, build logic must be broken down into discrete tasks with various inter-dependencies, and these typically map onto use cases such as those shown in Figure 1. Each task depends on zero or more other tasks, which must be performed beforehand, unless they are "up to date".

2.1 Make

I left the Make tool out of Table 1 because it's too primitive to be considered a proper modern build tool, but it's worth mentioning for some context. Makefile syntax provides a rudimentary way to express some build logic. A makefile consists of a series of rules (tasks), each of which has a target, a set of dependencies, and a series of commands (a "recipe"):

Listing 1:
A basic makefile (generally a file with the exact name "Makefile").

```
target1.o : prerequisite1.c prerequisite2.h
    command1
    command2

target2.o : prerequisite3.c prerequisite4.h
    command3
    command4

...
```

```
[user@pc]$ make
```

Make is one level above simply running the commands in a shell script.

If a target file does not exist, Make executes the rule to create it. Otherwise, to avoid unnecessary work, it checks timestamps to see if a target needs to be re-created. If you modify a prerequisite file (say, `prerequisite1.c`), its "mtime" (modification time) will be newer than that of the target (say, `target1.o`). Make will therefore rerun the commands in that rule to re-generate the target file. This may trigger other rules to be executed too, since the target for one rule may be a prerequisite for another.

However, in the modern age, Make just doesn't solve *enough* of the problem. It misses many opportunities for automating parts of the build process, and (if developers use it directly) it forces them to write a lot of "boilerplate"[b] build logic.

Makefiles are still used, especially for C/C++ projects, but only as a small piece of the overall build logic. They are usually auto-generated by some of the other build tools listed in Table 1 (like Autotools and CMake).

2.2 Ant

Ant is higher-level than Make. Whereas Make's rules operate file-by-file, Ant's tasks operate on (generally) collections of files at a time. You specify dependencies between tasks (e.g. task "test" might depend on task "compile"), rather than between individual input and output files.

Listing 2 shows a simple Ant build.xml file.

In Ant's XML file, each `<target>...</target>` represents one task, each having a name and a set of dependencies. The tasks in this case are "compile" (with no dependencies by default) and "jar" (which depends on compiling). Each `<property .../>` is essentially a variable, which can later be referred to with `${propertyname}`. Each target may involve the creation of several files.

[b] Boilerplate code is a term for extremely generic code, which could in theory be automatically generated. You're typically forced to write at least some boilerplate as a result of the tools, languages and libraries you use, but the less the better.

Listing 2:
Ant's build.xml file.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project name="AddressBook" basedir="." default="jar">
    <property name="src.dir"       value="src"/>
    <property name="build.dir"     value="build"/>
    <property name="dist.dir"      value="dist"/>
    <property name="main-class"    value="AddressBookApp"/>

    <target name="compile">
        <mkdir dir="${build.dir}"/>
        <javac srcdir="${src.dir}" destdir="${build.dir}"
            includeantruntime="false" />
    </target>

    <target name="jar" depends="compile">
        <mkdir dir="${dist.dir}"/>
        <jar destfile="${dist.dir}/${ant.project.name}.jar"
            basedir="${build.dir}">
            <manifest>
                <attribute name="Main-Class" value="${main-class}"/>
            </manifest>
        </jar>
    </target>
</project>
```

```
[user@pc]$ ant jar
```

2.3 Maven

Maven takes automation much further than Ant, and works on a **build-by-convention** principle. If your project directories are laid out according to standard Maven conventions, and you're not generally doing anything out of the ordinary, then almost no configuration is needed.

Maven also manages external dependencies. That is, you can specify software packages that your application depends on (for compilation), and Maven will automatically download and set them up locally as part of the build process. We'll discuss this further later on.

You configure a Maven project through a file called "pom.xml" (Project Object Model). Listing 3 shows a rough translation of the Ant example into Maven's XML notation.

The Maven example isn't much shorter than the Ant one, but, if you look closely, there are only actually five real pieces of information in our example pom.xml:

- The file format version: `<modelVersion>4.0.0</modelVersion>` (which is fixed);
- The ID of the development group: `<groupId>edu.curtin.cs</groupId>`;
- The name of the application: `<artifactId>addressbook</artifactId>`;
- The version of the application: `<version>1.0</version>`; and
- The class containing the `main()` method:
  `<mainClass>edu.curtin.cs.addressbook.AddressBookApp</mainClass>`.

We're *not* actually telling Maven anything about what commands to run or even where the files are located. Why? Because Maven adopts conventions. It *already knows*:

Listing 3: Maven's pom.xml file.

```
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>edu.curtin.cs</groupId>
    <artifactId>addressbook</artifactId>
    <version>1.0</version>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-jar-plugin</artifactId>
                <configuration>
                <archive>
                    <manifest>
                        <mainClass>edu.curtin.cs.addressbook.AddressBookApp
                        </mainClass>
                    </manifest>
                </archive>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

```
[user@pc]$ mvn install
```

- What the standard tasks are in a typical build process;
- Where to look for your source code: inside `src/main/java/`;
- Where to put the compiled code and jar file: inside `target/`; and
- What commands are required to make this happen.

This raises an important question: how do you actually write custom build logic when you need to? Maven might be smart, but it can't anticipate everything. Maven allows you to write plugins in Java, which you can then configure using the `<plugin>...</plugin>` element. (We've already used that in Listing 3 to configure the main class, because it so happens that creating a "jar" archive is handled by one of the standard Maven plugins.)

3 Jar Files and Manifests

Listing 2 and Listing 3 both mention something called the **manifest**, which is mostly-auto-generated and sits inside a .jar file.

Jar files are the conventional way of distributing compiled Java and JVM programs and libraries. A jar file is really (internally) just a zip file, and you can open it with any zip file viewer. Figure 2 shows what you might see if you do this.

Jar files are mainly just a bundle of .class files, because it's much simpler to distribute just one file. However, there is *metadata* that goes along with a software application or library that also needs to be bundled in; hence the manifest file `MANIFEST.MF`. This is an auto-generated text file containing a list of *attributes*, one per line, in a "`name: value`" format. For instance, here's the manifest file for a particular distribution of JUnit:

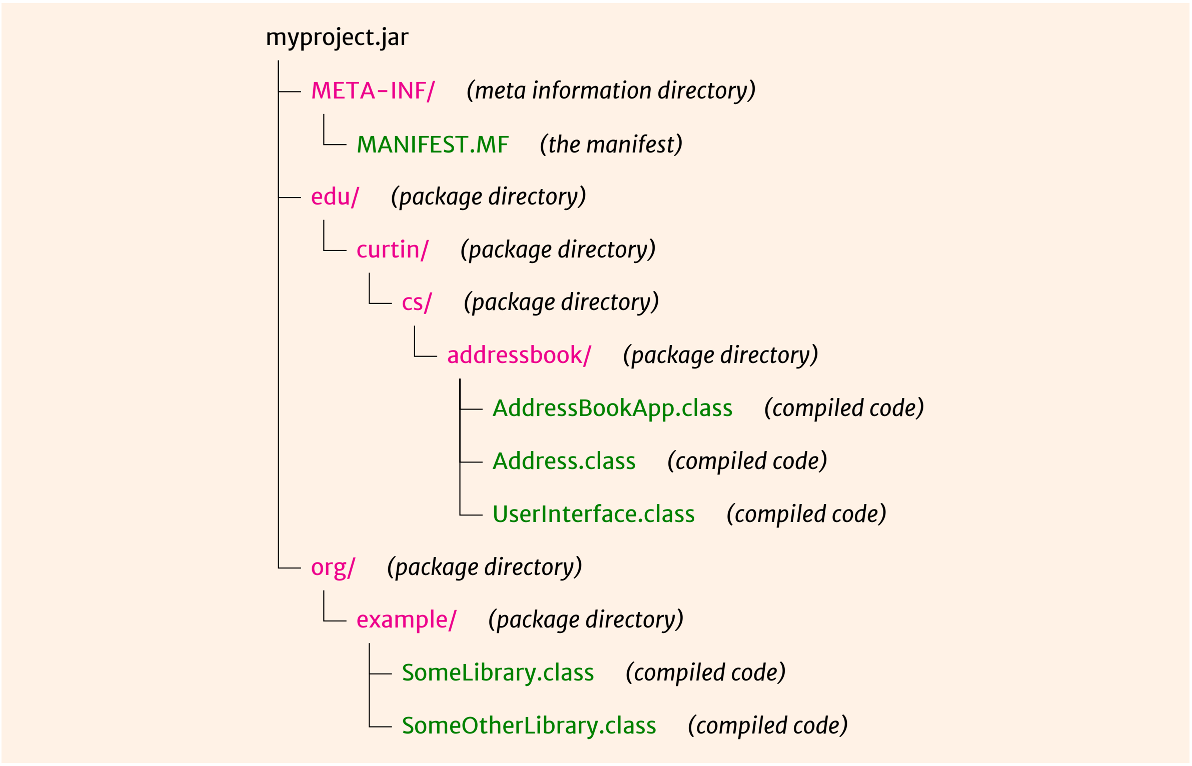
myproject.jar
META-INF/ (meta information directory)
MANIFEST.MF (the manifest)
edu/ (package directory)
curtin/ (package directory)
cs/ (package directory)
addressbook/ (package directory)
AddressBookApp.class (compiled code)
Address.class (compiled code)
UserInterface.class (compiled code)
org/ (package directory)
example/ (package directory)
SomeLibrary.class (compiled code)
SomeOtherLibrary.class (compiled code)

Figure 2: The inside of a jar file, which contains other files and directories, because it's really a zip file.

Listing 4: A `MANIFEST.MF` file from JUnit.

```
Manifest-Version: 1.0
Implementation-Vendor: JUnit
Implementation-Title: JUnit
Implementation-Version: 4.12
Implementation-Vendor-Id: junit
Built-By: jenkins
Build-Jdk: 1.6.0_45
Created-By: Apache Maven 3.0.4
Archiver-Version: Plexus Archiver
```

None of this is necessarily critical information, but in other cases there is one important attribute: **`Main-Class`**. To *run* a jar file (if it contains a runnable application), Java needs to know which .class file inside it contains the `main()` method. A manifest file for such an application might look like this:

Listing 5: A `MANIFEST.MF` file for a runnable jar file.

```
Manifest-Version: 1.0
Main-Class: edu.curtin.cs.addressbook.AddressBookApp
```

Build tools (Ant, Maven, Gradle) will create `MANIFEST.MF` for you (plus the jar file), but you must specify the value of `Main-Class`. Java then knows what to do when you run the following command:

```
[user@pc]$ java -jar myproject.jar
```

4 Gradle

We'll focus predominantly on the Gradle tool, as it lets us explore some of the more advanced concepts in build engineering.

Gradle too adopts a build-by-convention approach, and external dependency management. Unlike Ant and Maven, it does not use XML, but rather a fully-fledged programming language, which Gradle refers to as its "build language" or "domain-specific language".

To start with, for a *extremely shallow* understanding of using Gradle (version 5), you do the following:

1. Go to your project directory and run:

   ```
   [user@pc]$ gradle init
   ```

   - From the list, select "java-application" (if you're making a Java application).
   - Select the defaults for the other questions.
   - Gradle will now create some subdirectories, generate a basic configuration file, and also put itself in your project directory.
   - Only do this *once!*

2. Put all your production source code for your project in `src/main/java/`, and your test code in `src/test/java/`.
3. Build (compile, test and package) your application:

   ```
   [user@pc]$ ./gradlew build
   ```

4. Run it (for your own testing/debugging purposes) with:

   ```
   [user@pc]$ ./gradlew run
   ```

5. Distribute the zip file (found in `build/distributions/`) to your grateful users!

For a slightly less shallow understanding, you'll need to have a look at the "`build.gradle`" configuration file. Listing 6 shows an example comparable with the previous Ant and Maven examples. Gradle's Java plugin will (like Maven) fill in most of the required information with sensible defaults. Thus, a working build.gradle file can be mercifully short and to-the-point.

Listing 6: Gradle's build.gradle file.

```
plugins {
    id 'java'
}

jar {
    manifest {
        attributes 'Main-Class': 'edu.curtin.cs.addressbook.AddressBookApp'
    }
}
```

You can often *use* Gradle quite effectively just by adding the appropriate configuration settings to `build.gradle`. However, I want to give you a deeper understanding of what's really going on here. To do that, let's drill down a bit.

4.1 Gradle, Groovy and Domain-Specific Languages

There's a whole lecture on domain-specific languages (DSLs) later on – they're not just about Gradle – but we need to confront Gradle's one first. There are two variants of it, one based on Groovy, and one on Kotlin.

Groovy and Kotlin are both themselves general purpose languages (like Java), not domain specific ones, but they allow you to define things that *look and feel* like entirely new language features. And if it looks and feels like Gradle is making a new language (out of Groovy and Kotlin), then, from a certain point of view, that's exactly what it *is* doing.

"Normal" Groovy code looks a bit like Java, but with optional dynamic typing, and with a few other enhancements. It reuses the standard Java API (with more enhancements), compiles to standard JVM code, and can interoperate with any compiled Java classes. Kotlin is a more distinct language from Java, but we have to put it to one side because we can't cover everything!

Here are a few of Groovy's tricks:

- Writing "`{...}`" (apart from when it's defining the scope of a class, method, control statement, etc.) creates a *closure*. A closure is a method-like object that can be passed to other methods, which can then execute it at any point.

```
void myMethodA(Closure x)
{
    x()                 // Calling a closure
}
...
myMethodA( {            // Passing a closure
    println("Hello")
    println("world")
} )
```

  The nearest approximation in Java is a lambda expression or anonymous class, though Groovy's closures are always represented by the `Closure` class, and have additional features.

- You can omit the parentheses from a method call. Even if you do use parentheses, if the method takes a closure as its final parameter, the closure can appear *after* the closing parenthesis.

```
void myMethodB(int x, Closure y) {...}
...
// The following are all equivalent:
myMethodB(42, { println "Hello world" })
myMethodB 42, { println "Hello world" }
myMethodB(42) { println "Hello world" }
```

- You can create and pass `Map` values to methods by a simple "`key1: value1, key2: value2, ...`" syntax.

```
void myMethodC(Map x) {...}
...
myMethodC value: 42, str: "Hello world"
```

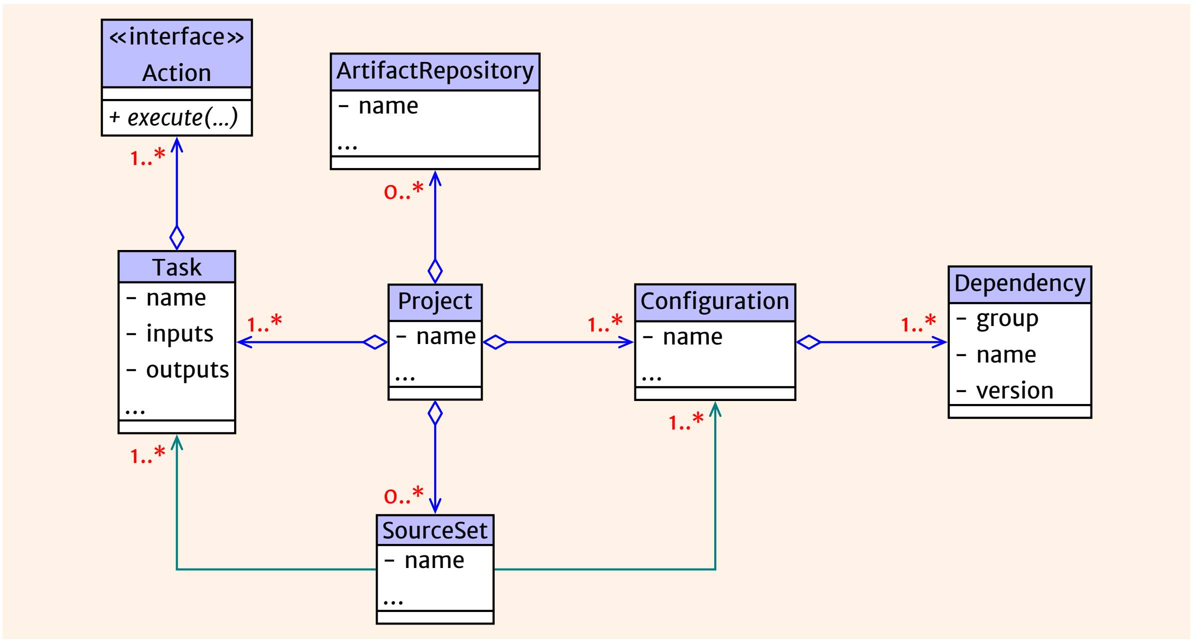

Figure 3: A highly simplified illustration of the Gradle class structure.

This helps shed light on what's going on at a language level in Listing 6:

- "`plugins`", "`jar`" and "`manifest`" are all methods that take a `Closure` parameter.
- "`id`" is a method accessible within `plugins{...}` that accepts a `String`.
- "`attributes`" is a method accessible within `manifest{...}` that accepts a `Map`.

4.2 The Project Model

The above discussion illustrates an important point: `build.gradle` is *real code*. It is not just a configuration file to be read, but a program to be run. So what does it *do?*

The purpose of `build.gradle` is to set up an object structure that represents all the information Gradle needs to build and manage the project. At the root of this structure is the `Project` object. This is where the methods `apply()` and `jar()` exist, for instance. The whole `build.gradle` file is executed as if it was a method of the project object. (You don't need to write "`project.`" in front of everything, but conceptually that's what's happening.)

From `Project`, we can define and modify a range of other objects:

- `Task` objects represent the units of actual work that need to be done; e.g., compiling, testing, packaging, etc.
- `SourceSet`s each represent a high-level bundle of source code. Conventionally you have a "main" source set for your production code, and a "test" source set for your unit test code. Large projects can find uses for others too.
- `Configuration`s represent sets of external dependencies for specific purposes. Applications typically rely on various third-party libraries, and these are made available as `Dependency` objects. But compilation, runtime and testing all tend to require (at least slightly) different sets of dependencies; hence we have multiple configurations.
- `Repository` objects represent the location (online or offline) where these external dependencies can be obtained.

Figure 3 shows a basic outline of the project model in UML, although be aware that the real structure is significantly more complex than this.

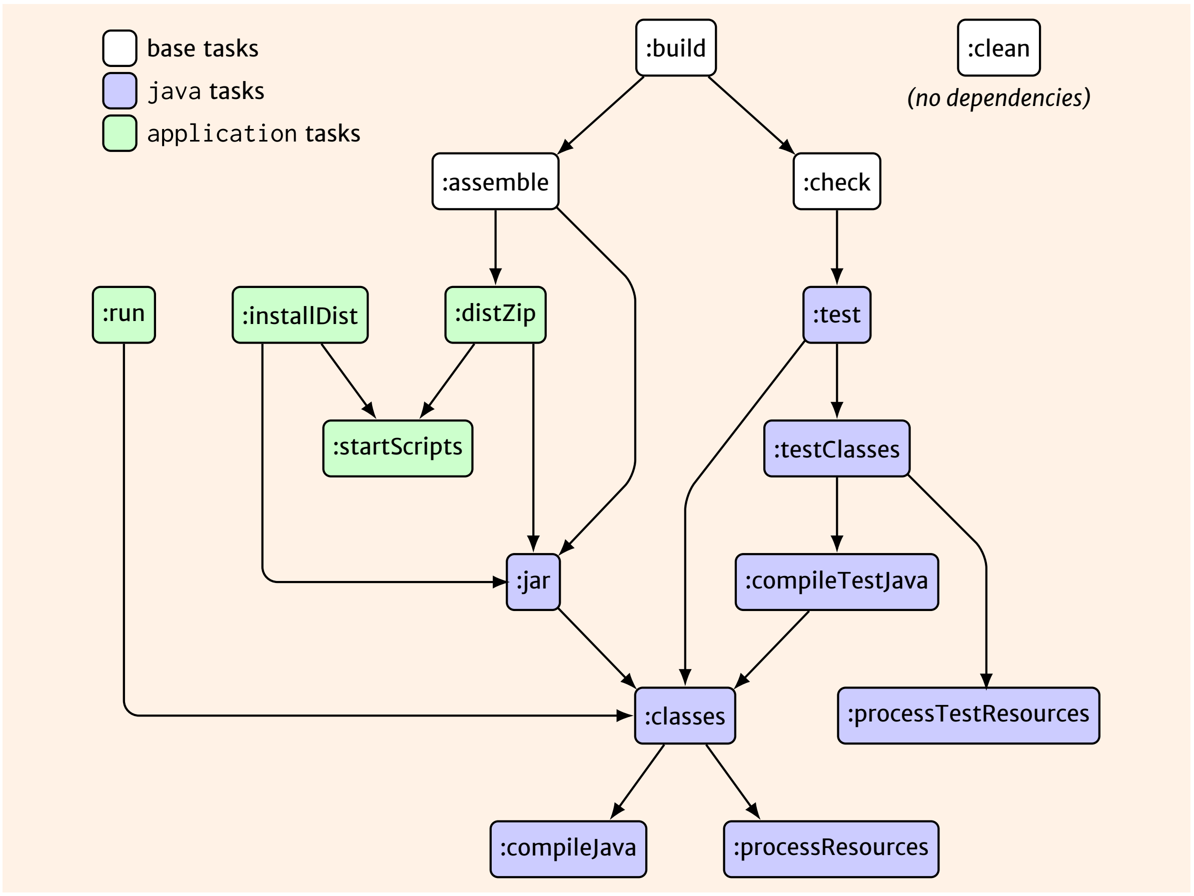

Figure 4: Gradle tasks and task dependencies created by the `java` and `application` plugins (as of Gradle 5.4).

The job of `build.gradle` – setting up all these objects – is called the *configuration phase* (not to be confused with the unfortunately-named `Configuration` objects!). Once done, Gradle moves onto the *execution phase*, where it actually performs useful work, including downloading dependencies from repositories, compiling source sets, and whatever else we specified during configuration.

In fact, there is also an *initialisation phase* that precedes configuration, responsible for creating the `Project` object itself. This becomes important when we have projects made up of **subprojects**, or projects within a project, each with its own individualised `build.gradle` file, and its own separate tasks, source sets, etc. There is a separate script file called `settings.gradle`, run as part of the initialisation phase, that tells Gradle what and where these subprojects are. More on that in Section 7.2.

Understanding the difference between these phases is crucial to understanding Gradle. It forces us to separate the "what" from the "how", and therefore write build logic at a higher level, more easily understood and maintained.

5 Tasks and Task Dependencies

Every build tool breaks up its work into **tasks**[c], which depend on other tasks. This is the most fundamental concept in build scripts.

In principle, `build.gradle` tells Gradle what tasks exist and what they depend on. In practice, `build.gradle` mostly delegates this responsibility to various "plugins". Figure 4 and Figure 5 show (most of) the standard conventional tasks that you get "for free" whenever you use the `java`, `application`, `cpp-application` and `cpp-unit-test` plugins.

[c] Different tools may have different names for them, though.

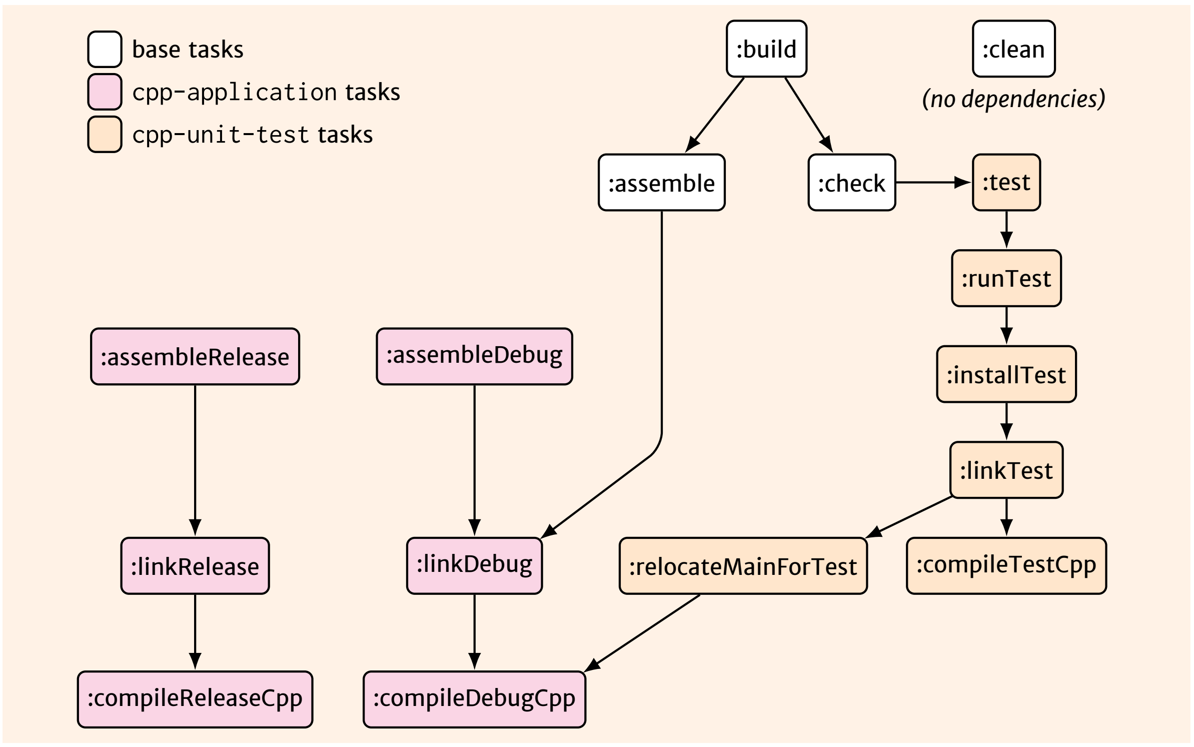

Figure 5: Gradle tasks for C/C++ plugins (as of Gradle 5.4).

When the build user runs Gradle, they specify a task. For instance typing "`./gradlew build`" (or "`./gradlew :build`") requests that Gradle runs the `build` task. More precisely, Gradle ensures that the specified task is **up-to-date**.

This first involves finding the task's dependencies, both direct and indirect. For instance, as you can see in Figure 4, the "build" task depends indirectly on most other tasks. Gradle then starts with the tasks that have no dependencies (such as "compileJava") and works upwards, running each task after its dependencies have been run. Eventually it will get to the task that the user requested, and then it will stop.

Once a given task has been run, it becomes "up-to-date". If you run Gradle again, and don't change anything relevant, that task will be skipped. This allows us to perform **incremental builds**, where Gradle only re-does the tasks that need re-doing. A typical day in the life of a software developer involves many small edits and incremental rebuilds.

Performing a full build is usually much slower, but if you *need* to do that, you can run the "clean" task first. That will delete all the results of any previously-run tasks, ensuring that nothing is up-to-date anymore. (Tasks also implicitly depend on the `build.gradle` file itself. If you change *that*, Gradle knows that the dependency structure itself could have changed.)

5.1 Defining and Customising Tasks

In moderately-complex projects, you may need to go beyond the standard conventions.

You can define a completely new task. This is a bit like defining a method or function, except that a task has a configuration part and an execution part. For instance:

Listing 7: Defining a Gradle task.

```
task myTask {
    println "Configuring myTask"
    ...

    doLast {
        println "Executing myTask"
        ...
    }
}
```

Given this, you can run:

```
[user@pc]$ ./gradlew myTask
```

It will print the two messages, one after the other. The main body of the task is the configuration part. In a plain-vanilla task like this, there's not much to configure, but we'll get back to it later.

The construct `doLast {...}` specifies what the task actually does in the execution phase. So does `doFirst`, in fact. `doFirst` and `doLast` can be used to add extra actions to the start and end of a sequence. Mostly you'll only need one action, but if you have several then they'll all get run in order. This is where you're most likely to see "ordinary code" in the `build.gradle` file – the `if` statements, loops, API calls, etc. (although these things are valid in most other places in the file too).

Once task is defined, you can connect it to other tasks via dependencies:

Listing 8: Creating Gradle task dependencies.

```
task otherTask { ... }
task myTask {
    ...
    dependsOn otherTask // Make 'myTask' depend on 'otherTask'
}

build.dependsOn myTask  // Make 'build' depend on 'myTask'
```

This creates two dependencies. First, we're saying that `myTask` cannot be run until `otherTask` is complete. Second, we're forcing `myTask` to run beforehand when we run `build`. Hence, if we do run `build`, the first thing that will actually happen is `otherTask`.

Even after tasks have been created, we can add to both their configuration and execution later on:

Listing 9: Modifying a task.

```
task myTask { ... } // Define the task initially here

...

myTask { // Modify the task
    println "Additional configuration"

    doLast {
        println "Additional execution action"
    }
}
```

This modification facility is probably more useful for the "conventional" tasks that are created for you, as in Figure 4. For instance:

Listing 10: Modifying conventional tasks.

```
compileJava {
    // Make the compiler output warning messages for unchecked generic types.
    options.compilerArgs.add "-Xlint:unchecked"
}

run {
    // When running the application via Gradle itself, handle console input.
    // (The default, a bit counter-intuitively, is to *disable* console input.)
    standardInput System.in
}
```

Having said this, how do you know what configuration options are available for each task?

5.2 Task Types

There are many subclasses of `Task`. Each `Task` subclass performs a particular kind of execution action (compile, run, zip, etc.), and has a particular set of configuration options for specifying what is being compiled, run, zipped, etc. and the fine details.

Most tasks are instances of one of the subclasses, and so they get this execution action "for free", without needing a separate `doFirst` or `doLast`.

Some tasks don't have an action at all, though, and are just bundles of dependencies that cause other tasks to run. The base tasks "`build`", "`assemble`" and "`check`" fall into this camp, as do "`classes`" and "`testClasses`". The type of these tasks is an internal Gradle class, and by definition isn't important, since there's nothing to configure.

The API documentation for a given `Task` subclass gives you its available configuration options. The reference documentation can unfortunately be quite terse, and Gradle also maintains two sets of reference documetation (with a given class often appearing in both):

- The "Gradle Build Language Reference" purports to be a high-level description (although not *that* high level!) of Gradle's language. You can see the task types (among other things) arranged down the left-hand-side of the page.
- The "Javadoc" reference is a lower-level description of each class, along the lines of Java's API documentation.

Table 2 shows some key `Task` subclasses.

Table 2: Some standard Gradle Task subclasses.

Task type	Standard tasks of this type	Explanation
`JavaCompile`	`compileJava`, `compileTestJava`	Invokes the Java compiler for a particular set of source files.
`JavaExec`	`run`	Executes a compiled Java application. Note: this is intended for debugging, not for normal use.
`CppCompile`	`compileReleaseCpp`, `compileDebugCpp`, `compileTestCpp`	Invokes a C++ compiler for a particular set of C/C++ source files.
`LinkExecutable`	`linkRelease`, `linkDebug`, `linkTest`	Invokes a native code linker for a set of compiled native (e.g. C/C++) code.
`Test`	`test`	Performs a series of unit tests, given compiled test code.
`Copy`	`processResources`, `processTestResources`	Copies a given set of files to a destination directory.
`Zip`	`distZip`	Creates a .zip file from a particular set of files.
`Jar`	`jar`	Creates a .jar file from a set of compiled .class files. Note: `Jar` is a subclass of `Zip`.
`Exec`	—	Executes an external command. Note: this may make your build environment platform-specific.
`Delete`	`clean`	Deletes a particular set of files.

You can create a task of an existing type by writing: "`task <name> (type: <type>) {...}`". For instance:

Listing 11: Defining a Gradle task of a pre-existing type.

```
// Copy all .class files from src/main/java into build/classes/java/main.
task copyPrecompiledClasses (type: Copy) {
    from 'src/main/java'
    include '**/*.class' // Only copy files matching this pattern.
                         // '**' means 'in any subdirectory'.
    into 'build/classes/java/main'
}

// Ensure this is done before we create the 'jar' file.
jar.dependsOn copyPrecompiledClasses
```

The `from`, `include` and `into` clauses are specific to the `Copy` task type, and specify what to copy and where. The practical use of the task in Listing 11 would be to allow pre-compiled .class files as part of your "source" code. The compiler only works with .java files, of course, and any other file types would normally be ignored.

6 External Dependencies

While there are obviously dependencies between *tasks*, in practice, most applications also depend on third-party software. We saw a couple of examples of this in lecture 3, where we used a class called `IOUtils`, which comes from a library called Apache Commons IO:

Listing 12: Depending on an external library (used to download a URL).

```
import org.apache.commons.io.IOUtils; // 3rd-party library class
...
URL myURL = ...;
String content = IOUtils.toString(myURL, "UTF-8"); // 3rd-party API call
System.out.println(content);
```

This code won't compile unless the compiler can find a working copy of Apache Commons IO, nor will it run unless the runtime environment can find it too. You can simply install it on your machine, but then so will the user, and you probably don't want to complicate the user's life.

Moreover, you could run into version conflicts. When using a library in your application, you must pick a *specific* version of it to use. Different versions have different features and different bugs, which you must adapt to. Typically you would pick the most recent "stable"/"release" version that is available at the time you write the code. Thereafter, you stick with that version even if new ones are released, unless there's a compelling reason to switch (and then you have a lot of testing to do).

If the installed version of the library is different from the one you actually want to use, it may not have all the classes and methods you expect it to have, or they may work differently. For instance, in Apache Commons IO version 2.0, `IOUtils` lacks the particular `toString()` method we want to call[d], so that version is useless to us!

If we take a step back, we realise that we're dealing with two different concepts:

- The **work environment** (for lack of a better term) consists of all the software you actually use on your machine, and the libaries required. This is generally subject to automatic or semi-automatic updating, and to administrative decisions taken by the IT managers in your organisation.
- The **build environment** consists of all the external dependencies you need to build your software. This is governed by the software development team, not the IT managers. The build environment must be kept constant, to ensure the reliability of the build process, unless and until where the development team decides otherwise.

We must keep these two environments separate, because they're based on completely different sets of decision-making. It's possible, for instance, that your work environment contains an application that requires Apache Commons IO 2.0, and breaks if a more recent version is used. (It's generally impractical to have multiple versions of the same library installed. How would our code know which version to use?)

Fortunately, build tools can help us maintain an isolated build environment, so that the versions of installed libraries are irrelevant.

Gradle, Maven and Ant/Ivy get you to specify your application's external dependencies – the library name, the exact version to use, and where to obtain it from. In the case of Gradle, we write the following:

[d] It doesn't have a `toString()` method that takes a `URL` parameter.

Listing 13: External dependencies in Gradle.

```
// This indicates where to download external dependencies from. You can use
// 'jcenter()' or 'mavenCentral()', or you can set up your own local or remote
// repository.
repositories {
    jcenter()
}

// This indicates what the external dependencies actually are, and which
// configuration they apply to.
dependencies {
    implementation 'commons-io:commons-io:2.5'

    // Can also be written as:
    // implementation group: 'commons-io', name: 'commons-io', version: '2.5'

    // You can also write "compile" instead of "implementation", but
    // "implementation" is now the preferred syntax.
}
```

Maven originally set the standard for this; hence, “commons.io” is the group ID, “commons.io” is also the name of the library, and “2.5” is the version required. The group ID generally identifies *who* the library's developers are, whereas the name identifies *what* the library is. They can be the same, as in this case, but are often different.

“Jcenter” is a comprehensive online repository of open source libraries[e], from which the build tool will attempt to locate the required 3rd-party software.

When building MyApp, Gradle will first download any required external dependencies and store them in its cache[f], if it hasn't already. Each external dependency itself may specify its own dependencies, called **transitive dependencies**. These are generally handled automatically (though there are rare cases where you may need to intervene to avoid conflicting transitive dependencies).

The benefit of this should be clear: developers can simply specify which libraries to use, and then use them. To change the version, you can simply change the version number in `build.gradle`, but otherwise it's guaranteed to remain the same.

So, why the word “`implementation`”? This is the name of the **configuration** that the dependency falls under. We mentioned configuraions previously in Section 4.2; each one represents a set of external dependencies for a specific purpose. There are a few conventional ones:

`implementation` represents external dependencies, typically libraries, needed for compiling and running your production code.

`runtimeOnly` – dependencies needed only at runtime. That is, the compiler doesn't need to know about them, but they need to be there when the application runs. This could include libraries that aren't *directly* called by the main application. It could also include non-code resources like fonts, icons or XML files.

`testImplementation` – dependencies needed specifically for compiling and running your

[e] The `jcentre()` repository refers to https://bintray.com/bintray/jcenter. There are others too, such as `mavenCentral()`.

[f] The cache is normally located in the directory `~/.gradle/`, which is outside the project directory. You may be working on several Gradle projects on the same machine, in which case they will share the same cache.

Listing 14: External dependencies for testing.

```
dependencies {
    implementation 'commons-io:commons-io:2.5'
    testImplementation 'junit:junit:4.12'
    testImplementation 'org.mockito:mockito-all:1.10.19'
}
```

```
[user@pc]$ ./gradlew test
```

tests. These might include unit test frameworks like JUnit or Spock, and associated libraries like Mockito. See Figure 14.

Note that any "`implementation`" dependencies apply automatically here too, since compiling and running tests is really a special case of compiling and running the whole application.

`testRuntimeOnly` – dependencies needed only while actually running your tests.

`api` – only relevant if your project is itself a library, using the "`java-library`" plugin. This configuration represents dependencies that your library needs, and which will "transitively" become dependencies of whichever *other* projects use your library.

(There is a difference between an `api` dependency and an `implementation` dependency. Even if your library depends on some other library, you don't necessarily need to pass on that dependency to applications that use your library. You can instead bundle the other library in with yours, so that it's "right there" when the application's build script downloads your library. However, there are times when you may not want to do this.)

6.1 The Gradle Wrapper

Gradle can perform another dependency trick that sounds paradoxical: it can find, download and run *itself*, alleviating you of the need to even have *it* installed.

The "Gradle wrapper" consists of the UNIX script file `gradlew`, the equivalent Windows script file `gradlew.bat`, and everything inside `gradle/wrapper`. These are normally created by running "`gradle init`" (or "`gradle wrapper`", if you prefer to set up your project manually). Running the `gradlew` script is more-or-less the same as running Gradle directly, except that it does the following:

- The wrapper searches the cache for the correct version of Gradle itself.
- If not there, the wrapper will download the correct Gradle version and store it in the cache (along with other external dependencies).
- The wrapper runs that specific version of Gradle, passing all the command-line parameters.

You can even upgrade the version of Gradle you're using:

```
[user@pc]$ ./gradlew wrapper --gradle-version=5.6
```

The Gradle wrapper is considered *part of your project* (and hence should be tracked by your version control repository). This way, you have more control over your build environment, ensuring that the operation of the build tool itself is unaffected by the software installed on the machine[g].

7 Project Structure

The physical directory structure for your project is important in its own right. Typically we have separate directories for the production code, test code, resources used by each, compiled classes, third-party libraries, and other things. You can in general determine whatever directory structure you like, but there are also conventions, and some build tools adhere to these by default.

For instance:

- `src/main/java/` contains any production Java code;
- `src/test/java/` contains any unit test Java code;
- `build/` (for Gradle) or `target/` (for Maven) contains all files created by the build process.

Gradle and Maven will use these locations as defaults unless you tell them otherwise. Thus, adhering to these conventions can simplify build logic.

7.1 Source Sets

In Gradle, all source code and other source things (XML files, icon images, etc.) that you create is conventionally located in `src/`.

Immediately inside `src/` are the **source set** directories. By default, Gradle gives you two of these: `src/main/`, your production code; and `src/test/`, your unit test code. (If you use Gradle to develop Android apps, you may notice another one: `src/androidTest/`, which contains *integration* test code.)

Within each source set directory, there is:

- A subdirectory for (broadly speaking) each language used, such as `java/`, `c/`, `cpp/`, `groovy/`, etc. In simple cases, you're probably only using one language, but in more complex cases you might use several.

  Each of these subdirectories contains the source code for the given language, though each language might *also* have its own directory structure that starts at this point.

  For instance, Java source code is typically stored in package directories. If you have a Java class called "`MyClass`" inside the package "`org.example.myapp`", then the path to this file will typically be "`src/main/java/org/example/myapp/MyClass.java`". The `src/main/java` part is the build system's directory structure. The rest is the responsibility of the language[h].

- A `resources` (or `res`) subdirectory, containing non-code files – XML files, images, etc. – that also form part of the source material.

Figure 6 shows how the directory structure for a project might look, given the two standard source sets (`main` and `test`), and assuming your application has both `Java` and `C++` code.

If you have a compelling reason to invent other source set(s), you can:

[g] One can be pedantic about this, of course. Even the Gradle wrapper requires that the Java Development Kit is installed.

[h] For Java source code, package directories are actually more convention than actual requirement. However, there is an equivalent package directory structure for the *compiled* code, and that *is* required, though also auto-generated.

Figure 6: An example directory layout for a project containing both Java and C++ code.

```
<root project directory>
├─ build/   (contains all output files, both final and temporary, from the build process)
│  └─ ...
├─ src/   (all source files)
│  ├─ main/   (source set for production code)
│  │  ├─ java/   (all Java production code)
│  │  │  └─ org/   (package directory)
│  │  │     └─ example/   (package directory)
│  │  │        └─ myapp/   (package directory)
│  │  │           ├─ MyClass.java   (source code)
│  │  │           └─ MyOtherClass.java   (source code)
│  │  ├─ cpp/   (all C++ production code)
│  │  │  ├─ my_native_code.cpp   (source code)
│  │  │  └─ my_other_native_code.cpp   (source code)
│  │  ├─ headers/   (all C++ production header files)
│  │  │  └─ my_native_code.h   (header file)
│  │  └─ resources/   (non-code resources)
│  │     ├─ myappicon.png   (image file)
│  │     └─ myguilayout.xml   (XML GUI layout)
│  │
│  └─ test/   (source set for unit test code)
│     ├─ java/   (all Java unit test code)
│     │  └─ org/   (package directory)
│     │     └─ example/   (package directory)
│     │        └─ myapp/   (package directory)
│     │           └─ MyTestClass.java   (source code)
│     └─ cpp/   (all C++ unit test code)
│        └─ my_native_test_code.cpp   (source code)
│
├─ gradle/   (contains the main Gradle wrapper program)
│  └─ ...
├─ gradlew   (Gradle wrapper script for Linux/UNIX)
├─ gradlew.bat   (Gradle wrapper script for Windows)
├─ build.gradle   (build script)
└─ settings.gradle   (initialisation file; see below)
```

Listing 15: Creating new source sets.

```
sourceSets {
    performanceTest
    proVersion
}
```

This will create two new source sets, stored at `src/performanceTest/` and `src/proVersion/`, with those directories having the same basic structure as `src/main/` and `src/test/`.

Additionally, for each new source set "xyz"[i], Gradle will automatically create the following:

- Three new tasks, for you to build parts of your source set: "`compileXyzJava`" (of type `JavaCompile`), "`processXyzResources`" (of type `Copy`), and "`xyzClasses`". There *won't* automatically be a dependency connecting these to the `build` task, although you can create that yourself if needed.
- Various new configurations, for you to supply dependencies to your source set; most notably: "`xyzImplementation`" and "`xyzRuntimeOnly`".

7.2 Subprojects

You can in principle use source sets to break up a project into parts. *However,* Gradle has a more general approach for doing this.

A **sub-project** is a project within a project. It is a complete project in its own right, with its own `build.gradle` file, its own source sets, and its own `build/` directory. However, it is located in a directory inside another "parent" project.

Subprojects can have any level of independence from one another (and even from the root project) that you desire. They can use completely different sets of plugins, tasks, configurations, repositories, dependencies, source sets, or programming languages. Or they can use the same ones!

Before you ask: yes, you can also have projects within projects within projects. You can have an arbitrary tree of subprojects, any number of levels deep. The rest of this discussion only looks at two levels – the root project and one level of subprojects. However, you should be able to extrapolate to more complex cases. You use a colon (":") to separate the names of parent and child projects; e.g. "`bigproject:innerproject:tinyproject`".

Subprojects are set up in Gradle's *initialisation* phase. This comes before the configuration phase, and also uses a different script file called `settings.gradle` (instead of `build.gradle`). `settings.gradle` is also written in Groovy, but generally has much simpler responsibilities than `build.gradle`. It might look like this:

Listing 16: Example `settings.gradle`.

```
// Defines the subprojects within this project
include 'app', 'web', 'desktop', 'server'

// Defines the name of the overall project
rootProject.name = 'myapp_deluxe'
```

Given this setup, our project directory should contain subdirectories `app/`, `web/`, `desktop/` and `server/`, along with the normal `src/` and `build/` directories seen so far. The four subproject directories themselves *also* contain their own individual `src/`, `build/`, etc.

[i] At least when you're using the `java` plugin.

Figure 7: An example directory layout for project containing several subprojects. Note: subproject names are all chosen by the development team.

```
<root project directory>
├─ build/ …  (root project output files (if any))
├─ src/   (root project source files (if any))
│   ├─ main/ …  (root project production code (if any))
│   └─ test/ …  (root project unit test code (if any))
│
├─ app/   (subproject directory; you choose the name)
│   ├─ build/ …  (subproject output files)
│   ├─ src/   (subproject source files)
│   │   ├─ main/ …  (subproject production code)
│   │   └─ test/ …  (subproject unit test code)
│   └─ build.gradle   (subproject build script)
│
├─ web/   (subproject directory)
│   ├─ build/ …  (subproject output files)
│   ├─ src/   (subproject source files)
│   │   ├─ main/ …  (subproject production code)
│   │   └─ test/ …  (subproject unit test code)
│   └─ build.gradle   (subproject build script)
│
├─ desktop/ …  (subproject directory)
├─ server/ …  (subproject directory)
│
├─ gradle/ …  (contains the main Gradle wrapper program)
├─ gradlew   (Gradle wrapper script for Linux/UNIX)
├─ gradlew.bat   (Gradle wrapper script for Windows)
├─ build.gradle   (root project build script)
└─ settings.gradle   (initialisation file specifying the subprojects)
```

To execute particular subproject tasks, we must run "`./gradlew :<subproject>:<task>`":

```
[user@pc]$ ./gradlew :desktop:build
[user@pc]$ ./gradlew :server:test
```

If you just give a task name by itself (with no subproject), using "`:`" means the difference between executing that task on the root project, or on every project in the hierarchy. For instance, the following builds the root project only, not its subprojects:

```
[user@pc]$ ./gradlew :build
```

By contrast, the following builds the root *and* all subprojects:

```
[user@pc]$ ./gradlew build
```

The root project's configuration does not apply to subprojects automatically. To configure subprojects, you must do one or more of the following:

- Setup each subproject's own `build.gradle` file, kept immediately within the subproject directory. This is the more usual approach, although it doesn't allow for reuse of declarations common across multiple subprojects.

  (Note: subprojects *do not* have a `settings.gradle` file, nor is there really any point in giving them each a copy of the Gradle wrapper.)

- Within the root project's `build.gradle`, use `subprojects {...}` or `allprojects {...}`:

  ```
  // In the root project's build.gradle...
  subprojects {
      // Declarations in here apply to all subprojects, in addition to their own
      // individual build.gradle files.

      // Current versions of Gradle have trouble with the 'plugins {...}' block
      // going here though. Instead, you can use a related construct:
      //     apply plugin: 'java'
  }

  allprojects {
      // Similar to 'subprojects {...}', but also applies to the root project.
  }
  ```

- For completeness, you can also configure a *specific* subproject from within the root project's `build.gradle` file:

  ```
  // In the root project's build.gradle...
  project(':mysubproject') {
      // Declarations in here apply only to the subproject 'mysubproject'.
  }
  ```

You can do any combination of the above, and the configuration options you give for each subproject will add up.

Subprojects can express dependencies on one another:

Listing 17: Subproject dependencies.

```
// desktop/build.gradle
dependencies {
    // Indicates that the 'desktop' subproject depends on the 'web' subproject.
    implementation project(':web')
}
```

When we want to build the "`desktop`" subproject, the above dependency causes "`web`" to be compiled first, and the resulting compiled code to be made available when compiling `desktop`.

However, Gradle does not allow the root project to depend on its own subprojects this way. If you find a need to do this, then instead simply move the root project's source code into a new subproject. (It is perfectly fine for the root project to not actually have any code in it.)

8 Further Reading

- Jessica Thornsby (2010), *Maven vs. Gradle vs. Ant*, https://jaxenter.com/maven-vs-gradle-vs-ant-101737.html
- Apache Software Foundation (2016), *Maven in 5 Minutes*, https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html.
- Gradle Inc. (2015), *Chapter 7. Dependency Management Basics*, Gradle User Guide, https://docs.gradle.org/current/userguide/artifact_dependencies_tutorial.html.
- Gradle Inc. (2015), *Chapter 14. Build Script Basics*, Gradle User Guide, https://docs.gradle.org/current/userguide/tutorial_using_tasks.html.
- Gradle Inc. (2015), *Chapter 24. Multi-project Builds*, Gradle User Guide, https://docs.gradle.org/current/userguide/multi_project_builds.html.
- Gradle Inc. (2015), *Chapter 44. Java Quickstart*, Gradle User Guide, https://docs.gradle.org/current/userguide/tutorial_java_projects.html.