Worksheet 4: Build Engineering

Updated: 19th September, 2019

1. Setting Up a Project

Obtain a copy of `diff.zip` from Blackboard. This contains a set of source files for a complete, if rather basic, "Diff" program. (It is inspired by the standard UNIX "`diff`" command for comparing two files and reporting the differences.)

The source files are as follows:

- **`Diff.java`** contains the core algorithm;
- **`DiffResult.java`** represents the result of the operation – a record of which parts of the two files are the same, and which parts are different;
- **`DiffTest.groovy`** is a set of unit tests for `Diff.java`, written in Groovy, using the Spock unit testing framework.
- **`DiffRunner.java`** contains a `main()` method, which takes two command-line parameters (filenames), invokes the diff operation, and outputs the results to the terminal, using coloured text to represent file differences;
- **`GUIDiffRunner.java`** also contains a `main()` method, but instead creates a GUI that asks the user to select two files, invokes the diff operation, and displays the result in a window.

  (It uses a Swing-based GUI. Swing has been superceded by JavaFX, but is still somewhat easier to use in this situation.)
- Two `.png` files contain image icons for use by the GUI.

All the Java/Groovy files are declared to be in the package "`edu.curtin.comp3003.diff`".

Your task here is to place these files inside a project directory structure, such that you can use Gradle to manage the build process. In other words, convert this collection of files into a functioning Gradle-based project. Use the notes to help you, but basically you need to do the following:

(a) Create a new project directory somewhere, and navigate to it in the terminal, whether in Linux, MacOS or Windows.

(b) Put the Gradle wrapper in it. You can do this by obtaining `gradle-wrapper.zip` from Blackboard and unzipping it.

  Alternatively, if you have Gradle installed, you can run this:

```
[user@pc]$ gradle wrapper --gradle-version=5.6.2
```

> **Note:** Doing this first differs slightly from the notes, but will help ensure everyone's experience is the same.
>
> From this point onwards, the installed version of Gradle (if any) is irrelevant. We'll be using the wrapper itself, which is inside the project directory. All further Gradle commands will begin with "`./gradlew`" (on Linux/MacOS) or just "`gradlew`" (on Windows).
>
> If you see "Permission denied" on Linux/MacOS when running "`./gradlew`", try instead running "`bash gradlew`".

(c) Initialise the project directory:

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

("`gradlew init`" on Windows.)

Gradle will ask you various questions. Tell it to create a Java application, using the Spock unit test framework.

(d) Put all the source files into their appropriate locations within "`src/`".

(The `init` command will have created hello-world versions of the production and test code. You can safely delete these.)

(e) Edit `build.gradle` and specify the correct name for the main class. In fact, there are two options for this. For now, consider `edu.curtin.comp3003.diff.GUIDiffRunner` to be the main class.

(f) Check whether the project builds by running:

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

In fact, you should initially get a unit test failure, due to a (deliberate and reasonably obvious) bug in `DiffTest.groovy`. This is just to show you what such an event looks like. Gradle will provide you with a link (that you can open in a web browser) to see all the details of the test failure.

Fix the offending line in `DiffTest.groovy`, and try building again.

(g) Check whether the project runs:

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

You should see a GUI window, as described above.

2. Sub-Projects, Jars and Distributions

Let's try to split up the Diff project into sub-projects.

(It's questionable whether we would do this for such a small project in the real world, but it's a good level of complexity for learning how sub-projects work.)

(a) Based on the notes, restructure the directory layout with the following sub-project directories:

`library` – contains `Diff.java`, `DiffResult.java` and `DiffTest.groovy`. This will not be an executable application, but just a library to be utilised by the other sub-projects.

`cli` – contains `DiffRunner.java`, and has a project-level dependency on `library`. This *is* an executable application, where `DiffRunner` is (of course) the main class.

`gui` – contains `GUIDiffRunner.java` and its image icon resources, also depends on `library`, and has `GUIDiffRunner` as its main class.

The root-level project doesn't need to contain any code, and its `build.gradle` file can be empty. However, you will need to edit `settings.gradle` to define the sub-projects. You also need to give each sub-project its own individualised `build.gradle` file, determining which parts of the original file need to belong to each sub-project.

(b) Now we'll check whether we got it right. First, get rid of any existing build files:

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

This will invoke the "clean" task on all three sub-projects.

(c) Build the CLI sub-project:

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

This should (if you set up the dependency properly) also compile "`library`".

(d) Take a look at the .jar files for both `library` and `cli`. You should be able to open them with any zip-file extractor tool, and you should be able to spot the appropriate .class files. Specifically:

`library/build/libs/library.jar` should contain `Diff.class`, `DiffResult.class` and `DiffResult$Entry.class` (the last one representing a nested class that appears within `DiffResult.java`).

`cli/build/libs/cli.jar` should contain `DiffRunner.class` *and not* any of the others.

Each .jar should also contain the manifest file `MANIFEST.MF`.

(e) There should *also* be a file called `cli/build/distributions/cli.zip`, which represents the complete distributable software; i.e., what you can give to your users.

It should contain four files: the same two `.jar` files from above, and two start-up script files, `bin/cli` (for Linux/MacOS) and `bin/cli.bat` (for Windows).

Check that it works:

(i) Extract/unzip it into some alternate location;

(ii) Find two convenient text files, which we'll call `file1.txt` and `file2.txt`;

(iii) Run the start-up script (depending on your OS):

```
[user@pc]$ ./cli file1.txt file2.txt
```

(Make sure to either copy `file1.txt` and `file2.txt` to the current directory, or specify their locations when running the command.)

If this works, and the files are not identical, you should see the red/green coloured Diff output.

> **Note:** Older versions of Windows may have difficulty displaying the colours, and may instead display the "ANSI escape codes" that are intended to produce the colours, but practically speaking this isn't important.

(f) Build the GUI sub-project:

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

Check that you can extract and run `gui/build/distributions/gui.zip`, in a similar fashion to before.

3. Custom Tasks

Say we care a lot about Diff's performance, and we'd like to create a custom task to measure it. Before we do, though, we need *another* custom task to generate test data. (That is, in order to measure Diff's performance, we need to run it. To run it, it needs something to compare.)

(a) We'll create some code to generate test data, as follows:

```groovy
class Generator     // Also available from Blackboard as 'Generator.groovy'
{
    private Random random = new Random();

    public String generateFile(String file, int lines, int lineWidth)
    {
        Writer w = new FileWriter(file);
        for(int i = 0; i < lines; i++)
        {
            for(int j = 0; j < lineWidth; j++)
            {
                w.write(random.nextInt(95) + 32);
            }
            w.write('\n');
        }
        w.close();
    }
}
```

This is Groovy code (and it's *almost* valid Java code too). You create a new `Generator` object, call `generateFile()`, passing in the name of the file to generate, the number of lines to generate, and the number of characters on each line. All the characters will be randomly chosen.

(There are infinitely-many alternative approaches to this, and you're welcome to improve on it if you like!)

But how do we integrate this with our build logic?

(i) Copy and paste the code directly into `build.gradle`. It is, after all, written in the same language! Put it in the *top-level* `build.gradle` file, for the purposes of this exercise.

(ii) Below the class definition, define a new task called "`generateTestData`", which performs the following action:

```
Generator gen = new Generator();
project.buildDir.mkdirs();
gen.generateFile("$project.buildDir/test1.txt", 5, 50);
gen.generateFile("$project.buildDir/test2.txt", 5, 50);
```

See the notes for the full syntax for creating a task.

The above code will generate two test files. In Groovy, "`$`" lets you embed variables in double-quoted strings (as mentioned previously), and in Gradle "`project.buildDir`" represents the full path to the "`build/`" directory. (It's nice to rely on variables, rather than repeating literal values, and it's also nice not have to make assumptions about the current working directory.)

The second line creates the `build/` directory if needed.

(iii) Run it:

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

And check the `build/` directory for `test1.txt` and `test2.txt`.

(iv) You might also be interested in seeing where it fits in with the existing tasks:

```
[user@pc]$ ./gradlew tasks --all
```

You should see a large listing of tasks, and you should find `generateTestData` in the "Other tasks" section. You can (if you wish) recategorise it and also give it a description, by adding the following lines to the task definition:

```
group = "Performance testing tasks"
description = "Creates new test data files"
```

(v) For convenience, enclose the `task` definition inside an `allprojects{...}` or `subprojects{...}` block. This will make it part of each sub-project. (The class definition must remain in the global scope, though.)

You can now execute any of the following:

```
[user@pc]$ ./gradlew :library:generateTestData
```

```
[user@pc]$ ./gradlew :cli:generateTestData
```

```
[user@pc]$ ./gradlew :gui:generateTestData
```

Each of these will generate a separate set of test data, located in each subproject's own `build/` directory.

Running "`./gradlew generateTestData`" will run all of the above.

(b) To do the actual performance checking, we'll focus back on the CLI sub-project. We'll create a modified/alternate version of the existing "`run`" task, which is based on the `JavaExec` task *type*.

A lot of the specifics of this are a beyond the scope of the notes, so I'll just present it to you here:

```
task performanceCheck(type: JavaExec) {
    // The 'classpath' indicates how to find all the Java classes that
    // form part of the application and its libraries.
    classpath = sourceSets.main.runtimeClasspath

    // We again need to say which class to actually run.
    main = mainClassName

    // We need to provide the "command-line" arguments -- our test data.
    args "$project.buildDir/test1.txt", "$project.buildDir/test2.txt"

    // Declare a variable to keep track of the time.
    def start

    doFirst {
        // *Before* we run main(), we make a note of the start time:
        start = System.currentTimeMillis()
        println "Starting test..."
    }

    doLast {
        // *After* we've run main(), calculate the elapsed time:
        println "Duration = ${System.currentTimeMillis() - start}ms"
    }
}

// Note: The JavaExec task type already defines an action: running a Java
// application. When we write doFirst{...} and doLast{...}, we're adding
// new actions onto the beginning and the end.
```

However, apart from copying and pasting this into `cli/build.gradle`, there are a

couple of other things for you to do:

(i) Set up the task dependencies! `performanceCheck` cannot be run until after both `generateTestData` and `jar` are complete. Additionally, we would like the pre-existing `check` task to cause `performanceCheck` to be run.

(ii) Run it:

```
[user@pc]$ ./gradlew :cli:clean
```

```
[user@pc]$ ./gradlew :cli:check
```

If all the dependencies have been set up properly, this should generate the test data, compile the code, and then do the performance check, printing out the duration in milliseconds.

End of Worksheet