IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Lab session – Build tools

Software Development Engineering – IDL

Correction

Objectives

The purpose of this lab session is to:

- familiarize you with more or less modern software construction processes;
- introduce you to build system technologies.

1 A first tool: `make`

In this section, you can use the documentation for the GNU version of `make` :
`https://www.gnu.org/software/make/manual/make.html`.
In the following, we will use the following *shell* commands:

- `touch filename` which allows you to create `filename` if it does not exist and otherwise update its date; this command allows you to simulate the creation or update of a file.
- `echo "text"` which allows you to display `text`; this allows you to monitor the triggering of rules.

These commands will allow you to easily simulate file production and the application of production rules.

Exercise 1

▷ **Question 1.1:**
Write a `Makefile` file that manages a system in which we have:

- **two input files: `a.in` and `b.in`,**

- **we know how to produce a `.out` file from a `.in` file, and**
- **we know how to produce `a_b` from `a.out` and `b.out`.**

```
#default target = first one or the one defined by .DEFAULT_GOAL

#.DEFAULT_GOAL := a_b
a.out: a.in
	touch a.out
	echo "producing a.out"
b.out: b.in
	touch b.out
	echo "producing b.out"
a_b: a.out b.out
	touch a_b
	echo "producing a_b"
```

▷ **Question 1.2:**
What happens if there are two production rules for the same file?

If both rules have an action part, `make` warns of the redefinition and only the last rule is applied. If only one of the rules defines an action, the dependencies are merged. Thus

```
a.out: a.in
	touch a.out
	echo "producing a.out"
a.out: d.in
```

is equivalent to

```
a.out: a.in d.in
	touch a.out
	echo "producing a.out"
```

Exercise 2

In rule actions, it is possible to retrieve the name of the file produced by `$@` (the name of the one that triggers the rule), the name of the first dependency by `$<`, the names of more recent dependencies by `$?`, and the names of all dependencies by `$^`.

It is also possible to define implicit rules for `make`. Such rules can use `%` either in the name of the product or in the names of its dependencies.

▷ **Question 2.1:**
Write an implicit rule to produce a *filename*`.out` file from any *filename*`.in` file.

```
%.out: %.in
```

```
	touch $@
	echo "producing $@ from $<"
```

▷ **Question 2.2:**
What happens if there is a concrete production rule for the same file in addition to the implicit rule?

If both rules have an action part, only the concrete rule is used. If the concrete rule does not define an action, the dependencies are merged. Thus, if

```
%.out: %.in
	touch $@
	echo "producing $@ from $<"
a.out: d.in
```

`a.out` is produced if `a.in` or `d.in` changes.

Exercise 3

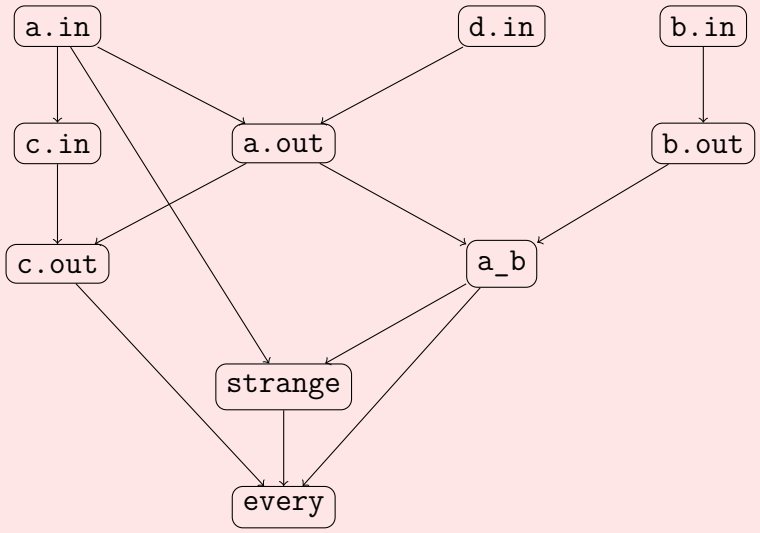
▷ **Provide the dependency graph for the `Makefile` below.**

```
a.out: a.in
	@touch $@
	@echo "producing $@ from $^ because of $?"
b.out: b.in
	@touch $@
	@echo "producing $@ from $^ because of $?"
a_b: a.out b.out
	@touch $@
	@echo "producing $@ from $^ because of $?"
a.out: d.in
c.in: a.in
	@touch $@
	@echo "producing $@ from $^ because of $?"
c.out: c.in a.out
	@touch $@
	@echo "producing $@ from $^ because of $?"
strange: a.in a_b
	@touch $@
	@echo "producing $@ from $^ because of $?"
every: c.out strange a_b
	@touch $@
	@echo "producing $@ from $^ because of $?"
.PHONY: clean
clean:
	rm *.out strange a_b every
```

```
.PHONY: clean
```

```
clean:
        ....
```

Allows you to declare a rule that does not produce a file and avoids a conflict with an existing file of the same name.



2 A second tool: `Gradle`

The goal here is to discover Gradle. To do this, download the dedicated archive from Moodle (`monstersandwizards.tar.gz`) anddecompress it. It includes two projects—`libgame` and `gnomegame`—which form a video game prototype. These exercises do not require you to look at or understand the Java code at any point.

This video game project is structured into two sub-projects[1]:

- libgame, which will contain the first part of the game content
- gnomegame: which will contain the game (the entry point: `Game.java`) as well as an extension `Gnome.Java`)

Unfortunately, when we look at the prototype's build system, we may doubt the innovative aspect of the project: it uses the *Ant* tool. . . , which you will be spared this year. In short, each subproject contains a `build.xml` file that specifies the various tasks involved in the software build process. It is not necessary to consult these files for the exercise. The following box summarizes the basic usage of *Ant* that you may want to remember:

[1]For the moment. This is obviously only the first step in conquering the cultural world with this highly disruptive and innovative game.

Since these files have the default name `build.xml`, simply type the following command in the directory containing the file:

```
$> ant
```

In this case, if a default target is selected, *Ant* attempts to execute it, along with all its dependencies. If any of them fail, the execution fails.
You can also execute a task defined in the `build.xml` file directly using the following command:

```
$> ant target_name
```

The following command provides help for the main *Ant* commands:

```
$> ant -h
```

To see all the tasks defined in a file, simply type the following command:

```
$> ant -p
```

In practice, typing the following commands into the terminal should be sufficient to compile and run the prototype:

```
$> cd monstersandwizards
$> cd libgame && ant
$> cd ../gnomegame && ant
```

If you open these `build.xml` files and take a look at how they work, you will understand the benefits of switching to a 21st-century software build system such as Gradle.

Gradle[2] is a multi-language build tool. It is embedded in the Groovy language[3] or, more recently, Kotlin. In this exercise, we will use the Groovy version.

The standard structure of a Gradle project is as follows:

`toto/`	the root of the project
`settings.gradle`	the list of subprojects
`build.gradle`	the configuration of the project construction
`gradle/`	the *wrapper* code
`gradlew`	the *wrapper* executable
`gradlew.bat`	the *wrapper* executable for Windows
`tata/`	the subprojects
`tutu/`	
`titi/`	

The *wrapper* allows a user to perform the build without having to install `gradle`. All that is required is to have Java installed.

The `settings.gradle` file defines subprojects using an `include 'tata', 'tutu', 'titi'` command. Thus, `gradle` will build these different subprojects[4]. It also allows you to define the name of the project: `rootProject.name = 'my game'`

[2] `https://gradle.org`
[3] `http://groovy-lang.org`
[4] The root project could also contain tasks to be performed, but in general it is just a container.

The `build.gradle` file contains the project build configuration. It generally consists of the project configuration and what is common to all subprojects. For example, below, we indicate that all subprojects are Java 8 projects and that external dependencies must be downloaded from the JCenter website[5].

```
subprojects {
  // all sub projects are java projects
  apply plugin: 'java'

    // ensure usage of a specific version of Java (1.8<Java<25)
    sourceCompatibility = '17'
    targetCompatibility = '17'

    // any external dependency must be downloaded from JCenter
    repositories {
      jcenter()
    }
}
```

Finally, each subproject has the following structure:

```
tata/                          the root of the subproject
      build.gradle             the build configuration of the subproject
      src/                     the sources of the subproject
         main/                 the code of the subproject
            java/              the Java code of the subproject
            resources/         the resources of the subproject
         test/                 the test code of the subproject
            java/              the Java test code of the subproject
            resources/         the test resources of the subproject
```

Gradle offers a tool for generating simple project structures with the command `gradle init`. This allows one to create various skeletons, depending on the choices made (build type, implementation language, language version, DSL of the scripts used, etc.).

Exercise 4

▷ **Based on the explanations above, build a standard Gradle project for the game that can be built.**

The composition of the project into two subprojects (*libgame* and *gnomegame*), as well as the dependency of *libgame* on the *Guava* library, will be taken into account. Note that *libgame* contains unit tests written in JUnit4, so this dependency will need to be integrated.

To use Gradle, you can choose to:

- install it on your machine using your package manager or via an installer.

[5] This is one of the standard websites for downloading dependencies.

- Use the `monstersandwizards-gradlewrapper.tar.gz` archive, which contains only the *wrapper*, saving you from having to install Gradle yourself. This *wrapper* uses Gradle-8.14.3.

To answer this question, you need to proceed in an orderly fashion:

1. retrieve (and decompress) the archive containing the code and the build system to be migrated;
2. ensure that Gradle is installed, install it, or use the wrapper we provide;
3. following the explanations in the Gradle section, create (or adapt) the structure of the project/subprojects;
4. create the `build.gradle` file at the root of the project (the one in the explanations above should suffice);
5. create the `settings.gradle` file adapted to the exercise at the root of the project;
6. create the `build.gradle` files for each sub-project;
7. test (using the `./gradlew run` command);
8. clean up any debris from the previous *build* system.

Following this procedure will likely yield a minimalist result (a few lines spread across several well-placed files).

Another option is to explore the possibilities of the `gradle init` command to generate one or more project skeletons. In this case, you can compare what this command generates depending on whether you initially choose an *Application*, a *Library*, or *Basic*. For the other choices, you will select the Java language (version 17, for example), the Groovy DSL, and generate the build without the new APIs (sould be the default choice, normally).

The structure:

```
|-- README
|-- build.gradle
|-- gnomegame
|   |-- build.gradle
|   `-- src
|       `-- main
|           `-- java
|               `-- game
|                   `-- monstersandwizards
|                       |-- Game.java
|                       `-- Gnome.java
|-- gradle
|   `-- wrapper
|       |-- gradle-wrapper.jar
|       `-- gradle-wrapper.properties
```

```
|-- gradlew
|-- gradlew.bat
|-- libgame
|   |-- build.gradle
|   `-- src
|       |-- main
|       |   `-- java
|       |       `-- game
|       |           `-- monstersandwizards
|       |               |-- GardenGnome.java
|       |               |-- Monster.java
|       |               |-- Person.java
|       |               |-- SuperPower.java
|       |               |-- Victim.java
|       |               |-- Warlock.java
|       |               `-- Wizard.java
|       `-- test
|           `-- java
|               `-- game
|                   `-- monstersandwizards
|                       |-- MonsterTest.java
|                       `-- PersonTest.java
`-- settings.gradle
```

The `settings.gradle` file:

```
rootProject.name = 'my game'
include 'libgame', 'gnomegame'
```

The `build.gradle` file of `gnomegame`:

```
dependencies {
    implementation project(':libgame')
}

apply plugin:'application'
mainClassName = "game.monstersandwizards.Game"
```

The `build.gradle` file of `libgame`:

```
dependencies {
    testImplementation 'junit:junit:4.+'
    implementation 'com.google.guava:guava:+'
}
```