



TP – Outils de construction

Ingénierie du développement logiciel – IDL

Objectifs

Ce TP a pour but de :

- vous familiariser avec les processus de construction de logiciels plus ou moins modernes ;
- vous faire découvrir des technologies de systèmes de *build*.

1 Un premier outil : make

Dans cette section, vous pourrez utiliser la documentation de la version GNU de `make` :

<https://www.gnu.org/software/make/manual/make.html>.

Dans la suite, on utilisera les commandes *shell* suivantes :

- `touch filename` qui permet de créer `filename` s'il n'existe pas et sinon de mettre à jour sa date ; cette commande permet de simuler la production ou la mise à jour d'un fichier.
- `echo "text"` qui permet d'afficher `text` ; on peut ainsi suivre le déclenchement des règles.

Ces commandes vont nous permettre de simuler simplement la production de fichier et l'application des règles de production.

Exercice 1

▷ Question 1.1 :

Écrire un fichier `Makefile` qui gère un système dans lequel on a :

- deux fichiers d'entrée : `a.in` et `b.in`,
- on sait produire un fichier `.out` à partir d'un fichier `.in` et
- on sait produire `a_b` à partir de `a.out` et `b.out`.

▷ Question 1.2 :

Que se passe t'il s'il existe deux règles de production pour un même fichier ?

Exercice 2

Dans les actions d'une règle, il est possible de récupérer le nom du fichier produit par \$@ (le nom de celui qui déclenche la règle), celui de la première des dépendances par \$<, celui des dépendances plus récentes par \$? et celui de toutes les dépendances par \$^.

Il est aussi possible de définir des règles implicites pour `make`. De telles règles, peuvent utiliser % soit dans le nom du produit soit dans ceux de ses dépendances.

▷ **Question 2.1 :**

Écrire une règle implicite pour produire un fichier *filename.out* à partir de n'importe quel fichier *filename.in*.

▷ **Question 2.2 :**

Que se passe t'il s'il existe une règles concrète de production pour un même fichier en plus de la règle implicite ?

Exercice 3

▷ **Donner le graphe de dépendance du Makefile ci-dessous.**

```
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
```

2 Un deuxième outil : Gradle

L'objectif est ici de découvrir l'outil **Gradle**. Pour cela, récupérer l'archive dédiée sur Moodle (`monstersandwizards.tar.gz`) et la décompresser. Elle comprend deux projets – `libgame` et `gnomegame` – qui forment une ébauche de jeu vidéo. Ces exercices n'exigent à aucun moment de regarder et comprendre le code Java.

Ce projet de jeu vidéo est structuré en deux sous-projets¹ :

- `libgame` qui contient la première partie du contenu du jeu
- `gnomegame` qui contient le jeu (le point d'entrée : `Game.java`) ainsi qu'une extension (`Gnome.java`)

Malheureusement, lorsque l'on regarde le système de *build* du prototype, on peut douter de l'aspect innovant du projet : en effet, il utilise l'outil *Ant*... qui vous est épargné cette année. Pour faire court, chaque sous-projet contient un fichier `build.xml` qui spécifie les différentes tâches du processus de construction du logiciel. Il n'est pas utile d'aller consulter ces fichiers pour l'exercice. L'encadré suivant résume l'usage basique de *Ant* que vous pouvez éventuellement retenir :

Comme ces fichiers portent le nom par défaut `build.xml`, il suffit de taper la commande suivante dans le répertoire contenant le fichier :

```
$> ant
```

Dans ce cas, si une cible par défaut est sélectionnée, *Ant* tente de l'exécuter, ainsi que toutes ses dépendances. S'il l'une échoue, l'exécution échoue.

On peut aussi exécuter directement une tâche définie dans le fichier `build.xml` par la commande suivante :

```
$> ant target_name
```

La commande suivante fournit de l'aide pour les principales commandes *Ant* :

```
$> ant -h
```

Pour connaître toutes les tâches définies dans un fichier, il suffit de taper la commande suivante :

```
$> ant -p
```

Concrètement, taper les commandes suivantes dans le terminal devrait suffire à compiler et exécuter le prototype :

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

Si vous ouvrez ces fichiers `build.xml` et vous penchez sur leur fonctionnement, vous comprendrez l'intérêt de passer à un système de construction de logiciel du XXI^{ème} siècle tel que Gradle.

Gradle² est un outil de construction multi-langages. Il est embarqué dans le langage Groovy³ ou plus récemment Kotlin. Dans cet exercice, nous allons utiliser la version Groovy.

La structure standard d'un projet Gradle est la suivante :

1. Pour le moment. Ce n'est évidemment que la première étape de la conquête du monde culturel avec ce jeu hautement *disruptif* et *innovant*.

2. <https://gradle.org>

3. <http://groovy-lang.org>

toto/	la racine du projet
settings.gradle	la liste des sous-projets
build.gradle	la configuration de la construction du projet
gradle/	le code du <i>wrapper</i>
gradlew	l'exécutable du <i>wrapper</i>
gradlew.bat	l'exécutable du <i>wrapper</i> pour Windows
tata/	
tutu/	
titi/	
	les sous-projets

Le *wrapper* permet à un utilisateur de réaliser la construction sans avoir à installer `gradle`. Il suffit d'avoir Java installé.

Le fichier `settings.gradle` contient la liste des sous-projets sous la forme d'une commande `include 'tata', 'tutu', 'titi'`. Ainsi, `gradle` va construire ces différents sous-projets⁴. Il permet également de définir le nom du projet : `rootProject.name = 'my game'`

Le fichier `build.gradle` contient la configuration de la construction du projet. Il consiste, en général, en la configuration du projet et ce qui est commun à tous les sous-projets. Par exemple, ci-dessous, nous indiquons que tous les sous-projets sont des projets Java 8 et que les dépendances externes doivent être téléchargée depuis le site de JCenter⁵.

```
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()
    }
}
```

Enfin chaque sous-projet à la structure suivante :

-
4. Le projet racine pourrait aussi contenir des tâches à réaliser mais en général il est juste un conteneur.
 5. Il s'agit de l'un des sites standard de téléchargement de dépendances.

tata/	la racine du sous-projet
build.gradle	la configuration de la construction du sous-projet
src/	les sources du sous-projet
main/	le code du sous-projet
java/	le code Java du sous-projet
resources/	les ressources du sous-projet
test/	le code de test du sous-projet
java/	le code Java de test du sous-projet
resources/	les ressources de test du sous-projet

Gradle offre un outil de génération de structure de projet simple avec la commande `gradle init`. Cela permet de créer divers squelettes, en fonction des choix effectués (type de *build*, langage d'implémentation, version du langage, DSL des scripts utilisés, etc.).

Exercice 4

▷ À partir des explications ci-dessus, construisez un projet Gradle standard du jeu que l'on peut construire.

On prendra bien en compte la composition du projet en deux sous-projets (*libgame* et *gnomegame*), ainsi que la dépendance de *libgame* à la bibliothèque *Guava*. Notez que *libgame* contient des test unitaires écrits en JUnit4, il faudra donc intégrer cette dépendance.

Pour utiliser Gradle, vous pouvez, au choix :

- l'installer sur votre machine avec votre système de paquets ou via un installateur.
- partir de l'archive `monstersandwizards-gradlewrapper.tar.gz` qui ne contient que le *wrapper*, ce qui vous évite d'installer Gradle par vous-même. Ce *wrapper* utilise Gradle-8.14.3.

Pour répondre à cette question, il faut procéder de manière ordonnée :

1. récupérer (et décompresser) l'archive contenant le code et le système de *build* à migrer ;
2. s'assurer que Gradle est installé, l'installer ou utiliser le *wrapper* que nous fournissons ;
3. en suivant les explications du sujet sur Gradle, créer (ou adapter) la structure du projet/des sous-projets ;
4. créer le fichier `build.gradle` à la racine du projet (celui des explications plus haut devrait suffire) ;
5. créer le fichier `settings.gradle` adapté à l'exercice à la racine du projet ;
6. créer les fichiers `build.gradle` de chaque sous-projet ;
7. tester (via la commande `./gradlew run`) ;
8. nettoyer les éventuelles scories du précédent système de *build*.

En suivant cette procédure le résultat sera probablement minimaliste (quelques lignes réparties dans différents fichiers bien placés).

Une autre option consiste à explorer les possibilités de la commande `gradle init` pour générer un ou plusieurs squelettes de projets. Dans ce cas, vous pouvez comparer ce que cette commande génère selon que vous fassiez le choix initial d'une *Application*, d'une *Library* ou *Basic*. Pour les

autres choix, vous sélectionnerez le langage Java (version 17 par exemple), le DSL Groovy, et générerez le *build* sans les nouvelles API (choix par défaut, normalement).