Assemblage de logiciels

Guy Tremblay
Professeur

Département d'informatique
UQAM
http://www.labunix.uqam.ca/~tremblay

6 octobre 2016

Contenu

1 Introduction/motivation

2 Qu’est-ce que l’assemblage de logiciels ?

3 Quelques outils d’assemblage de logiciels
- L’outil `make` sur Unix
- L’outil `Ant`
- L’outil `Maven`
- L’outil `Rake`

4 Conclusion

1. Introduction/motivation

Rappel : Parmi les premières choses à faire, quand on développe du code de façon professionnelle. . .

«*[Y]ou need to get the development infrastructure environment in order. That means adopting (or improving) the fundamental Starter Kit practices :*

- *Version control*
- *Unit Testing*
- *Build automation*

Version control needs to come before anything else. It's the first bit of infrastructure we set up on any project.»

«*Practices of an Agile Developer—Working in the Real World*», Subramaniam & Hunt, 2006.

Quelques *tips* de Hunt & Thomas

«The Pragmatic Programmer»

11. *DRY—Don't Repeat Yourself*
    Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

21. *Use the Power of Command Shells*
    Use the shell when graphical user interfaces don't cut it.

61. *Don't Use Manual Procedures*
    A shell script or batch file will execute the same instructions, in the same order, time after time.

4. *Script builds from day one*

   You have a problem if you do anything *by hand in your build or packaging process. [. . .] [It's] important that the build—in all its complexity—can be launched with a single command.*

2. Qu’est-ce que l’assemblage de logiciels ?

Qu’est-ce que l’assemblage d’un logiciel ?

A **build** *converts source code into a runnable program. Depending on the computer language and environment, this may mean you’re compiling source code and/or bundling images and other resources together as needed.*

The scripts, programs, and technologies used by your compile and resource building are all combined to create the **build system***.*

Source: «*Ship it !*»

Qu’est-ce que l’assemblage d’un logiciel ?

Historically, build has often referred either to the process of converting source code files into standalone software artifact(s) that can be run on a computer, or the result of doing so.

The process of building a computer program is usually managed by a build tool, a program that coordinates and controls other programs. [. . .] The build utility needs to compile and link the various files, in the correct order. If the source code in a particular file has not changed then it may not need to be recompiled [. . .].

Source: `https://en.wikipedia.org/wiki/Software_build`

Il faut distinguer entre *composants sources* et *composants dérivés*

Composants sources (*source components*)

Les composants qui sont — et doivent être — créés manuellement par les développeurs.

Composants dérivés (*derived components*)

Les composants qui peuvent être été créés automatiquement par la machine, sans l'intervention explicite des développeurs.

Source: «*Essential Open Source Toolset*», Zeller & Krinke

Il faut distinguer entre *composants sources* et *composants dérivés*

Composants sources (*source components*)

Les composants qui sont — et doivent être — créés manuellement par les développeurs.

Exemple pour programmes C : Fichiers `*.[hc]`, `Makefile`.

Composants dérivés (*derived components*)

Les composants qui peuvent être été créés automatiquement par la machine, sans l'intervention explicite des développeurs.

Exemple pour programmes C : Fichiers `*.o`, exécutables.

Source: «*Essential Open Source Toolset*», Zeller & Krinke

Il faut distinguer entre *composants sources* et *composants dérivés*

Remarque importante en lien avec le contrôle du code source :

- Seuls les fichiers pour les composants sources doivent être mis dans le système de contrôle du code source.

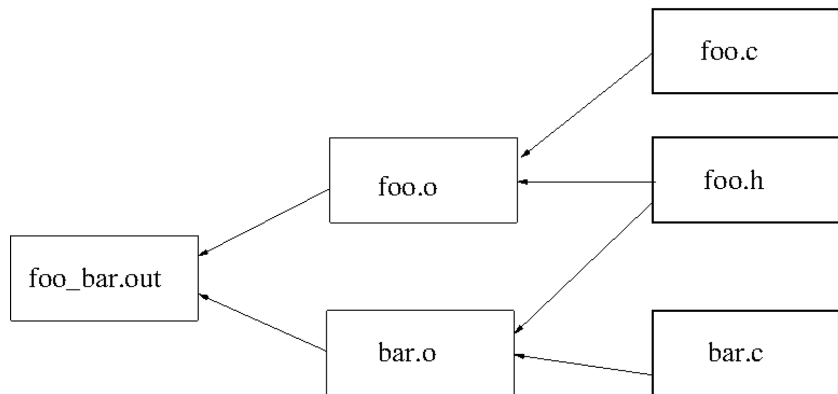
  $\Rightarrow$ Les fichiers pour les composants dérivés ne devraient pas être mis dans le système de contrôle du code source.

Le rôle d'un outil d'assemblage de logiciels

Le rôle d'un outil d'assemblagee de logiciels est d'assurer qu'un logiciel soit assemblé...

- à partir des bons composants sources
- toujours de la bonne façon
- sans intervention humaine
- rapidement — par ex., en regénérant le nombre minimal de composants dérivés
  $\Rightarrow$ on ne recompile que le strict nécessaire

Tous les outils d'assemblage reposent sur un modèle du logiciel à construire $\Rightarrow$ Graphe des dépendances



Le processus d'assemblage repose sur l'analyse du graphe de dépendances

Soit un composant dérivé A qui dépend de $A_1, A_2, \ldots, A_n$.

Alors, le composant A doit être regénéré si une des conditions suivantes s'applique :

1.
2.
3.

Source: «*Essential Open Source Toolset*», Zeller & Krinke

Le processus d'assemblage repose sur l'analyse du graphe de dépendances

Soit un composant dérivé A qui dépend de $A_1, A_2, \ldots, A_n$.

Alors, le composant A doit être regénéré si une des conditions suivantes s'applique :

1. A n'existe pas
2. un des composants A_i a été modifié
3. un des composants A_i doit être regénéré

Source: «*Essential Open Source Toolset*», Zeller & Krinke

Le processus d'assemblage repose sur l'analyse du graphe de dépendances

Soit un composant dérivé A qui dépend de $A_1, A_2, \ldots, A_n$.

Alors, le composant A doit être regénéré si une des conditions suivantes s'applique :

1. A n'existe pas
2. un des composants A_i a été modifié
3. un des composants A_i doit être regénéré

Donc : il s'agit donc d'un processus récursif !

Source: «*Essential Open Source Toolset*», Zeller & Krinke

3. Quelques outils d’assemblage de logiciels

3.1 L’outil `make` sur Unix

Qu’est-ce que `Make` ?

In software development, Make is a utility that automatically builds executable programs and libraries from source code by reading files called makefiles which specify how to derive the target program. [...]

Besides building programs, Make can be used to manage any project where some files must be updated automatically from others whenever the others change.

Source: *https://en.wikipedia.org/wiki/Make_(software)*

Un exemple simple : Compilation et exécution d'un fichier `hello.c`

```
$ cat hello.c
#include <stdio.h>

int main()
{
  printf( "Hello, World!\n" );
  return 0;
}
```

Un exemple simple : Compilation et exécution d’un fichier `hello.c`

```
$ cat makefile
default: run # Premiere cible = cible par defaut. Le nom n’a pas d’importance

run: hello
	./hello

hello: hello.c
	gcc -o hello hello.c

clean:
	rm -f hello hello.o
```

Un exemple illustrant que l'outil `Make` ne s'applique pas uniquement à la compilation de «programmes»

Dans mon éditeur de texte (`emacs`), la clé «F12» est associée à la macro `save-and-make` :

- Sauve le contenu du *buffer* dans le fichier.
- Lance l'exécution de `make` dans le répertoire courant.

Deux cas :

- Traitement de texte avec LaTeX : `make` regénère le PDF
  $\Rightarrow$ met à jour la version affichée à l'écran.
- Écriture de code : `make` lance l'exécution des tests.

Un exemple illustrant que l'outil Make ne s'applique pas uniquement à la compilation de «programmes»

```
$ cat makefile
MAIN=build

default: $(MAIN)

$(MAIN).pdf: $(MAIN).tex macros.tex biblio/*tex
        pdflatex $(MAIN)

slides_etudiants: clean
        $(HOME)/cmd/avec-notes.sh
        make $(MAIN).pdf
        scp $(MAIN).pdf tremblay@zeta:public_html/MGL7460/Mate

slides_prof:
        $(HOME)/cmd/sans-notes.sh
        make $(MAIN).pdf
        scp $(MAIN).pdf tremblay@zeta:public_html/maison/COURS
```

Ce que fait `Make` : Il analyse le graphe de dépendances entre tâches décrit par un `Makefile`

Forme générale d'une règle

$target_1$ $target_2$ ... $target_n$: $source_1$ ... $source_m$
$command_1$
$command_1$
...
$command_k$

`makefile` = description d'une ou plusieurs tâches (règles), où pour chaque tâche on a ...

- le nom de la tâche = la cible
- les tâches préalables = dépendances entre tâches
- les commandes (*shell*) à exécuter pour la tâche

Ce que fait `Make` : Il analyse le graphe de dépendances entre tâches décrit par un `Makefile` ⋆

- il analyse le graphe de dépendances des tâches
- il détermine les tâches qui sont absolument nécessaires pour construire la cible désirée
- il exécute les commandes associées aux tâches identifiées

Ce que fait `Make` : Il analyse le graphe de dépendances entre tâches décrit par un `Makefile`

Processus utilisé :

- Fouille en profondeur (*depth-first search*) du graphe de dépendances ⇒ algorithme récursif

- Utilise la date de dernière modification d'un fichier pour déterminer si le fichier a été modifié :
  - si *A* dépend de *B*
    et la date de dernière modification de *B* est après celle de *A*
    alors *B* a été modifié

Contenu d'un `Makefile`

- Des commentaires :

```
# Un commentaire debute par un diese.
```

- Des déclarations de variables :

```
MAIN = build
```

- Des cibles et leurs dépendances (possiblement avec variables) :

```
$(MAIN).pdf: $(MAIN).tex macros.tex biblio/*tex
```

- Les commandes à exécuter pour une cible :

```
pdflatex $(MAIN)
```

Contenu d'un `Makefile` et caractère de tabulation

Remarque importante : Une ligne indiquant une commande à exécuter doit débuter par un caractère de tabulation !

> *«The fact that commands have to be indented by tabs (and not, for example, blank spaces !) is the biggest stumbling block for beginners wanting to create a Makefile. Stuart Feldman [(le concepteur de* `make`*)] himself apparently encountered the same problem a few days after finalizing the original version of* MAKE *;* MAKE *had already been too widely distributed by then for him to be able to correct this design error.»*

Source*: «Essential Open Source Toolset», Zeller & Krinke*

Lancement de la commande `make`

- On identifie la cible principale :
  - Commande «`make`» :
    - Utilise la première cible définie dans le fichier.
  - Commande «`make foo`» :
    - utilise la cible «`foo`».
  - Commande «`make foo` VAR=bar» :
    - utilise la cible «`foo`» et définit la variable `VAR` comme étant `bar`.
- Récursivement, toutes les cibles dont les dépendances ne sont pas satisfaites sont exécutées
- Ne réexécute que ce qui est nécessaire

Lancement de la commande `make`

```
$ make -n
pdflatex build

$ make -n build.pdf
pdflatex build

$ make -n MAIN=tests-unitaires
pdflatex tests-unitaires
```

Note : «`make -n`» montre ce qui serait exécuté, mais sans l’exécuter réellement !

Contenu d'un Makefile (suite)

- Des règles de dépendances implicites :

```
# Regle pour compilation des fichiers .c
CC = gcc
CFLAGS = -ggdb -Wall -O

# Regle implicite
.c.o:
	$(CC) -c $(CFLAGS) $<
```

Variables spéciales dans les règles implicites :

- `$<` : Le premier préalable
- `$@` : La cible

Contenu d’un Makefile (suite)

- Une autre forme, plus générale, pour des règles de dépendances implicites (GNU Make) :

```
# Regle pour compilation des fichiers .c
CC = gcc
CFLAGS = -ggdb -Wall -O

# Regle implicite avec dependances additionnelles.
%.o: %.c definitions.h
	$(CC) -c $(CFLAGS) -o $@ $<
```

Patron spécial de règles implicites :

- `%` : patron pour une chaine de caractères arbitraire

Écriture d’un Makefile (1)

- À la main $\Rightarrow$ on détermine, à la main, les dépendances, et on écrit, à la main, chaque série de commandes

Écriture d'un Makefile (1)

- À la main $\Rightarrow$ on détermine, à la main, les dépendances, et on écrit, à la main, chaque série de commandes

Désavantages :

- Cibles omises
  $\Rightarrow$ Éléments manquants dans le logiciel déployé
- Dépendances omises
  $\Rightarrow$ Recompilations inutiles

Écriture d'un Makefile (2)

- À l'aide du compilateur, pour générer les dépendances :

```
$ ls *[hc]
biblio.c             MiniCUnit.c
bool.h               MiniCUnit.h
communications.c     outils.c
communications.h     outils.h
configuration.h      sequences.c
emprunts.c           sequences.h
emprunts.h           testCommunications.c
gererArguments.c     testEmprunts.c
gererArguments.h     testGererArguments.c
gererBD.c            testGererBD.c
gererBD.h            testGererErreurs.c
gererErreurs.c       testMiniCUnit.c
gererErreurs.h       testOutils.c
```

Écriture d'un Makefile (2)

- À l'aide du compilateur, pour générer les dépendances :

```
$ gcc -MM *.[hc]
biblio.o: biblio.c gererArguments.h gererBD.h emprunts.h\
          bool.h gererErreurs.h communications.h\
          configuration.h outils.h
bool.o: bool.h
communications.o: communications.c outils.h bool.h communi
...
sequences.o: sequences.c bool.h sequences.h
...
testGererErreurs.o: testGererErreurs.c configuration.h\
                    gererErreurs.h bool.h MiniCUnit.h
testMiniCUnit.o: testMiniCUnit.c MiniCUnit.h bool.h
testOutils.o: testOutils.c outils.h bool.h MiniCUnit.h
```

Écriture d'un Makefile (3)

- À l'aide d'outils tels que (GNU) `autoconf` et `automake`, qui génèrent un fichier `Makefile.in` (cibles, dépendances et règles) à partir d'un fichier de configuration minimale

Pour une introduction, voir notamment
`http://mij.oltrelinux.com/devel/autoconf-automake`

Quelques cibles standards

- `all` : Compile le programme en entier
- `test` : Teste le programme
- `install` : Compile le programme et l’installe
- `uninstall` : Désinstalle le programme
- `clean` : Supprime les fichiers temporaires créés par `Make`

3.2 L’outil `Ant`

Qu'est-ce que `Ant` ?

Ant est un logiciel créé par la fondation Apache qui vise à automatiser les opérations répétitives du développement de logiciel telles que la compilation, la génération de documents (Javadoc) ou l'archivage au format JAR, à l'instar des logiciels Make.

Ant est écrit en Java et son nom est un acronyme pour «Another neat tool» (un autre outil chouette). Il est principalement utilisé pour automatiser la construction de projets en langage Java, mais il peut être utilisé pour tout autre type d'automatisation dans n'importe quel langage.

Source*: `https://fr.wikipedia.org/wiki/Apache_Ant`*

Un désavantage de `Make` : problèmes de (non-)portabilité entre divers environnements

- `Makefile` (environnement Unix) :

```
clean:
        rm -rf classes
```

- `Makefile` (environnement Windows) :

```
clean:
        rmdir /S /Q classes
```

- `Ant` (n'importe quel environnement) :

```
<delete dir="classes"/>
```

Un désavantage de `Make` : on doit spécifier explicitement les commandes pour les cibles

- Si on veut créer une cible, on doit exécuter certaines commandes bien spécifiques
- Des variables et règles implicites peuvent être utilisées pour avoir des commandes flexibles, mais ce n'est pas l'outil lui-même qui sait comment générer les cibles.

Un avantage de `Ant` : l'outil fait lui-même le lien entre les cibles et les commandes

Avec `Ant`, on spécifie des tâches plutôt que des commandes.

`Ant` détermine quelles commandes exécuter, et ce pour de nombreuses tâches prédéfinies :

- Compilation Java
- Création d'archives (jar, zip, rpm)
- Génération de la documentation (Javadoc)
- Gestion des fichiers
- Exécution des tests unitaires
- Etc.

Avec `Ant`, les tâches et leurs dépendances sont définies dans un fichier `build.xml`

Un fichier `build.xml` contient :

- Une description d'un projet — nom et cible par défaut
- Une ou plusieurs cibles — nom et (possiblement) dépendances
- Une ou plusieurs tâches associées à chacun des cibles — donc des activités à faire pour générer la cible

Avec `Ant`, les tâches et leurs dépendances sont définies dans un fichier `build.xml`

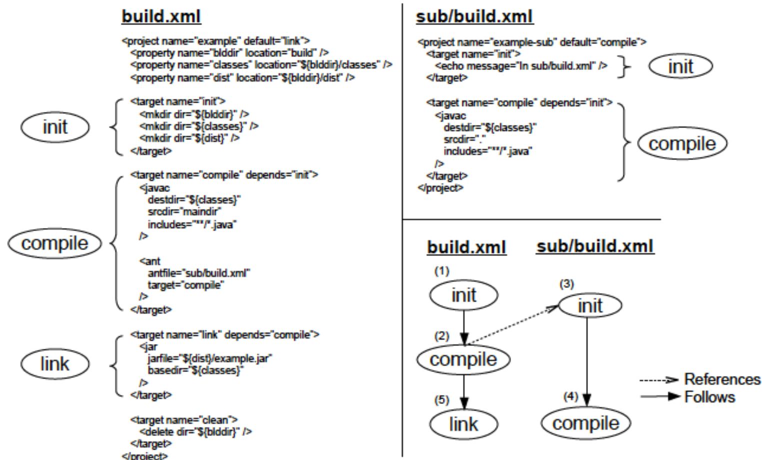


Fig. 1 Example ANT build.xml files (left, top-right) and the resulting build graph (bottom-right). The build graph has a depth of 2 (i.e., “compile” in build.xml references “init” in sub/build.xml) and a length of 5 (i.e., execute (1), (2), (3), (4), then (5)).

Source: «*The Evolution of Java Build Systems*», McIntosh et al.

Un désavantage de `Ant` = Les tâches et leurs dépendances sont définies en XML ☹

```
$ cat build.xml
<project default="hello">
  <target name="hello">
    <javac srcdir="."
           includeantruntime="false"
           destdir="."/>
  </target>

  <target name="clean">
    <delete>
      <fileset dir="." includes="*.class"/>
    </delete>
  </target>
</project>
```

Le lancement des tâches est semblable à `Make`

```
$ ls *.java
Hello.java

$ ant
Buildfile: build.xml

hello:
    [javac] Compiling 1 source file to
    /zusagers/tremblay/MGL7460/Materiel/Exemples

BUILD SUCCESSFUL
Total time: 4 seconds

$ ant hello
Buildfile: build.xml

hello:

BUILD SUCCESSFUL
Total time: 0 seconds
```

Le comportement de `Ant` peut être étendu en définissant de nouvelles tâches

- `Ant` est écrit en Java
- `Ant` définit une classe `Task` dont on peut hériter pour définir une nouvelle sorte de tâche

Désavantages de `Ant`

- Scripts XML ☹
- Seule notion plus abstraite = tâche… d'assez bas niveau
- Langage déclaratif, mais avec quelques *tags* impératifs

  ⇒ modèle/métaphore pas clair/claire
- Difficile de réutiliser des tâches et scripts — mécanisme limitée d'abstraction et réutilisation

  ⇒ Pas inhabituel d'avoir un fichier `build.xml` contenant des milliers de lignes ☹

Source: «*Continuous Delivery*», Humble & Farley

3.3 L’outil `Maven`

Qu'est-ce que `Maven` ?

Apache Maven est un outil pour la gestion et l'automatisation de production des projets logiciels Java en général et Java EE en particulier. L'objectif recherché est comparable au système Make sous Unix : produire un logiciel à partir de ses sources, en optimisant les tâches réalisées à cette fin et en garantissant le bon ordre de fabrication.

Il est semblable à l'outil Ant, mais fournit des moyens de configuration plus simples, eux aussi basés sur le format XML.

Source*: `https://fr.wikipedia.org/wiki/Apache_Maven`*

Objectif de Maven = Standardisation du processus d'assemblage

Maven was created with build process standardization in mind, since many Java projects of the Apache foundation had to re-implement the same ANT targets and tasks over and over again. These common build activities were consolidated into the dedicated concept of a Maven Build Lifecycle*.*

Source: «*The Evolution of Java Build Systems*», McIntosh et al.

L'approche «*Convention over Configuration*»

Convention over configuration is a software development approach geared toward developing programs according to typical programming conventions, versus programmer defined configurations. It enables quick and simple software creation while maintaining base software requirements.

Convention over configuration is also known as coding by convention.

Source: https ://www.techopedia.com/definition/27478/convention-over-configuration

Quelques exemples bien connus de systèmes fondés sur l'approche «*Convention over configuration*»

- Ruby on Rails
- Spring Framework
- Apache Maven

Maven repose sur une approche «*Convention over configuration*»

Convention over configuration is a simple concept. Systems, libraries, and frameworks should assume reasonable defaults. Without requiring unnecessary configuration, systems should "just work". [. . .] Hooks [can be] provided for you to override these default, assumed names if the need arises, but, in most cases, you will find that using the framework-supplied defaults results in a faster project execution.

Maven incorporates this concept by providing sensible default behavior for projects.

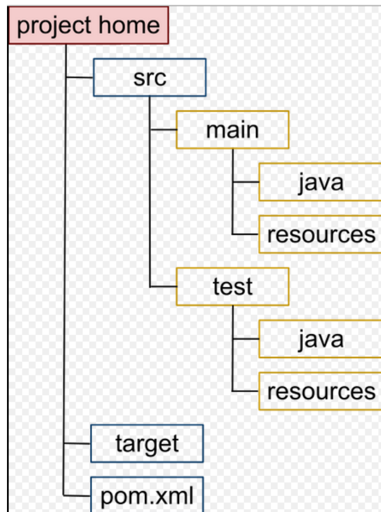
Source: http://books.sonatype.com/mvnref-book/reference/installation-sect-conventionConfiguration.html

Avantage de «Convention over configuration» dans le contexte de Maven

Maven attempts to remove the large amount of boilerplate found in Ant files by having a more complex domain that makes many assumptions about the way your Java project is laid out. This principle of favoring convention over configuration means that, so long as your project conforms to the structure dictated by Maven, it will perform almost any build, deploy, test, and release task you can imagine with a single command, without having to write more than a few lines of XML. That includes creating a website for your project which hosts your application's Javadoc by default.

Source*: «Continuous Delivery», Humble & Farley*

Organisation typique d’un projet utilisant `Maven`



Source: `https://en.wikipedia.org/wiki/Convention_over_configuration`

Création d’un projet avec mvn

```
$ ls

$ mvn archetype:generate\
-DgroupId=ca.uqam.app\
-DartifactId=mon-app\
-DarchetypeArtifactId=maven-archetype-quickstart\
-DinteractiveMode=false
```

```
[INFO] Scanning for projects...
[INFO]
[INFO]
------------------------------------------------------------------------
[INFO] Building Maven Stub Project (No POM) 1
[INFO]
------------------------------------------------------------------------
...
[INFO] Generating project in Batch mode
...
[INFO] BUILD SUCCESS
[INFO]
------------------------------------------------------------------------
[INFO] Total time: 25.016s
[INFO] Finished at: Wed Sep 23 18:33:08 EDT 2015
[INFO] Final Memory: 14M/106M
[INFO]
------------------------------------------------------------------------
```

Création d’un projet avec `mvn`

```
$ ls
mon-app/

$ tree mon-app
mon-app
|-- pom.xml
`-- src
    |-- main
    |   `-- java
    |       `-- ca
    |           `-- uqam
    |               `-- app
    |                   `-- App.java
    `-- test
        `-- java
            `-- ca
                `-- uqam
                    `-- app
                        `-- AppTest.java
```

Création d’un projet avec mvn

```
$ cat pom.xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation=’’http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/maven-v4_0_0.xsd’’>
  <modelVersion>4.0.0</modelVersion>
  <groupId>ca.uqam.app</groupId>
  <artifactId>mon-app</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>mon-app</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Un *Build Lifecycle* vise à regrouper des activités communes typiques d'un processus d'assemblage

Build lifecycle

«A [build] lifecycle is composed of one or more sequential phases in a fixed order (imposed by the Maven designers).»

Phase

«Each phase may contain zero or more sequential goals.»

Source: «*The Evolution of Java Build Systems*», McIntosh et al.

Trois *build lifecycles* sont automatique prédéfinis

- `default` : déploiement

- `clean` : nettoyage

- `site` : création de la documentation

Le `default` *build lifecycle* est composé de différentes phases — différentes cibles

1. `validate` : *vérifie si correct et information disponible*
2. `compile` : *compile le code source code*
3. `test` : *exécute les tests unitaires*
4. `package` : *crée version distribuable, e.g., JAR*
5. `integration-test` : *déploie pour des tests d'intégration*
6. `verify` : *vérifie la validité et qualité du paquetage*
7. `install` : *installe dans un dépôt local approprié*
8. `deploy` : *installe dans un dépot distant*

Source:

`https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html`

Avantage de `Maven`

Modèle complexe de cycle de vie et de développement

$\Rightarrow$

Très puissant, qui supporte (oblige) une bonne discipline de développement

Désavantages de `Maven`

- Difficile — pour ne pas dire impossible — d'utiliser `Maven` si le projet ne respecte pas la structure prévue

- Comme `Ant`, utilise un DSL externe en XML
  $\Rightarrow$ Difficile à étendre

- La configuration par défaut est *self-updating*
  $\Rightarrow$ Les *builds* peuvent être non reproductibles

Source: «*Continuous Delivery*», Humble & Farley

3.4 L’outil `Rake`

Qu'est-ce que `rake` ?

Rake is a build language, similar in purpose to make and ant. Like make and ant it's a Domain Specific Language, unlike those two it's an internal DSL programmed in the Ruby language.

Source : *http ://martinfowler.com/articles/rake.html*

[Rake] came about as an experiment to see if Make's functionality could be easily reproduced by creating an internal DSL in Ruby. The answer was "yes," and Rake was born.

Source : *«Continuous Delivery», Humble & Farley*

Note : Voir à la fin de cette section pour un `MiniRake` !

Caractéristiques de Rake

- Comme `Make` : `Rake` se fonde uniquement sur les notions de tâches et de dépendances.
- Comme `Make` : `Rake`, bien qu'écrit en Ruby, peut être utilisé avec n'importe quel langage
- À la différence de `Make` : un script `Rake` est un programme Ruby comme n'importe quel autre (DSL interne) !
- À la différence de `Make` : les commandes (e.g., manipulation de fichiers) d'un `Rakefile` peuvent être écrites de façon indépendante de la plateforme

Exemple = compilation d’un fichier `.c` : Avec `Make`

```
default: run

# Execution du programme.
run: hello
	./hello

# Generation de l’executable.
hello: hello.c
	gcc -o hello hello.c

# Nettoyage.
clean:
	rm -f hello hello.o
```

Exemple = compilation d'un fichier .c : Avec Rake ...

```
task :default => :run

desc 'Execution du programme'
task :run => 'hello' do
  sh %{./hello}
end

desc 'Generation de l\'executable'
file 'hello' => ['hello.c'] do
  sh %{gcc -o hello hello.c}
end

desc 'Nettoyage'
task :clean do
  rm_f 'hello hello.o'
end
```

Exemple = compilation d'un fichier `.c` : Avec `Rake`

Est-ce que cela ne vous fait pas penser à quelque chose ?

```
task :default => :run

desc 'Execution du programme'
task :run => 'hello' do
  sh %{./hello}
end

desc 'Generation de l\'executable'
file 'hello' => ['hello.c'] do
  sh %{gcc -o hello hello.c}
end

desc 'Nettoyage'
task :clean do
  rm_f 'hello hello.o'
end
```

Quelques exemples d'exécution

Construction de la cible par défaut

```
$ rake
gcc -o hello hello.c
./hello
Hello, World!
```

Construction d'une cible déjà dérivée

```
$ rake hello
gcc -o hello hello.c
$ rake hello
$
```

Quelques exemples d'exécution

Liste des tâches disponibles, avec description

```
$ rake -T
rake clean  # Nettoyage
rake hello  # Generation de l'executable
rake run    # Execution du programme
```

Liste des tâches, avec ou sans description

```
$ rake -T -A
rake clean    # Nettoyage
rake default  #
rake hello    # Generation de l'executable
rake run      # Execution du programme
```

Quelques exemples d’exécution

Dry run

```
$ rake -n
** Invoke default (first_time)
** Invoke run (first_time)
** Invoke hello (first_time, not_needed)
** Invoke hello.c (first_time, not_needed)
** Execute (dry run) run
** Execute (dry run) default
```

L’utilisation de règles implicites permet de réduire le nombre de tâches explicites

Règle style Make

```
rule '.o' => '.c' do |task|
  sh %{gcc -o #{task.name} #{task.source}}
end
```

L'utilisation de règles implicites permet de réduire le nombre de tâches explicites

Règle avec *pattern-matching*

```
rule /\.o$/ => '.c' do |task|
  sh %{gcc -o #{task.name} #{task.source}}
end
```

L'utilisation de règles implicites permet de réduire le nombre de tâches explicites

Règle avec détection dynamique

```
articles = Rake::FileList.new( '**/*.md' ) do |files|
  files.exclude( '~*' )
  files.exclude( /^temp.+\// )
  files.exclude { |file| File.zero? file }
  ...
end

detect_files = lambda do |task|
  articles.detect { |article| article.ext == task.ext }
end

rule '.html' => detect_file do |task|
   sh %{...}
end
```

Source: Adapté/simplifié de «*Rake Task Management Essentials*», Koleshko

Avantages/Désavantages de `Rake`

Avantages

- DSL Interne ⇒ Puissant et expressif ☺
- Disponible sur toute plateforme où Ruby est disponible

  Notamment, `JRuby` fonctionne sur toute plateforme avec une JVM (machine virtuelle Java)

Désavantages

- DSL Interne ⇒ Il faut connaitre (un peu) Ruby ☹
- Il faut installer Ruby et divers *gems*
- `JRuby` lent à démarrer (JVM !) ⇒ un appel à `rake` exécuté avec `JRuby` est plus lent qu'un appel à `make`

4. Conclusion

Tableau synthétique comparant `make`, `ANT` et `Maven`

Table 1 Build concepts.

	`make`	ANT	Maven
Build specifications	`Makefiles`, *.mk	build.xml	pom.xml
Unit of build activity	Build commands inside build rule.	ANT tasks inside ANT targets.	Goals bound to a particular phase.
Unit of abstraction	Build rules specifying how to build particular build target (file or abstraction).	Custom ANT targets specifying how to perform a conceptual build activity.	Standardized Maven phases specifying how to perform a conceptual build activity.
Dependency management	A target is only rebuilt if one or more of its dependent targets have been rebuilt.	A target is only rebuilt if one or more of its dependent targets have been rebuilt.	A fixed sequence of phases.

Source: «*The Evolution of Java Build Systems*», McIntosh et al.

La maintenance des scripts d’assemblage peut demander un effort non-négligeable ☹

Des études ont montré que la conception et l’entretien du système de *build* pouvaient entrainer des surcoûts (*overhead*) de l’ordre de 10 %.

La maintenance des scripts d'assemblage peut demander un effort non-négligeable ☹

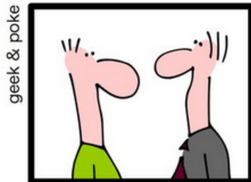
Des études ont montré que la conception et l'entretien du système de *build* pouvaient entrainer des surcoûts (*overhead*) de l'ordre de 10 %.

- Ajouter de nouvelles règles pour traiter des nouveaux fichiers sources
- Ajuster la configuration du compilateur ou de l'éditeur de liens
- …

Source: Article cité dans «*The Evolution of Java Build Systems*», McIntosh et al.

Bref, l’étape de `build` est cruciale

Source : `geek-and-poke.com`



HOW TO BECOME INVALUABLE

Tous les outils d'assemblage utilisent un DSL

DSL = *Domain Specific Language*

A computer programming language of limited expressiveness focused on a particular domain.

Source*: «*Domain-Specific Languages*», Fowler*

Tous les outils d'assemblage utilisent un DSL

DSL Externe

«An external DSL is a completely separate language, for which you [need] a full parser.»

DSL Interne

«An internal DSL is an idiomatic way of using a general-purpose language.»

Outils de *build* et DSL

Outil	**Type de DSL**	**Syntaxe**
Make	Externe	*Ad hoc*
Ant	Externe	XML
Maven	Externe	XML
Rake	Interne	Ruby

Références

J. Humble and D. Farley.
Continuous Delivery—Reliable Software Releases Through Build, Test, and Deployment Automation.
Addison-Wesley, 2011.

A. Koleshko.
Rake Task Management Essentials.
Packt Publishing, 2014.

S. Mcintosh, B. Adams, and A.E. Hassan.
The evolution of Java build systems.
Empirical Softw. Eng., 17(4-5) :578–608, August 2012.

J. Rasmusson.
The Agile Samurai—How Agile Masters Deliver Great Software.
The Pragmatic Bookshelf, 2010.

V. Subramaniam and A. Hunt.
Practices of an Agile Developer—Working in the Real World.
The Pragmatic Bookshelf, 2006.

A. Zeller and J. Krinke.
Essential Open Source Toolset.
John Wiley & Sons, Ltd, Chichester, UK, 2005.

A. `MiniRake` : Un `rake` simplifié (à venir)