Construction automatique...

Philippe Collet

Licence 3 MIAGE – S6

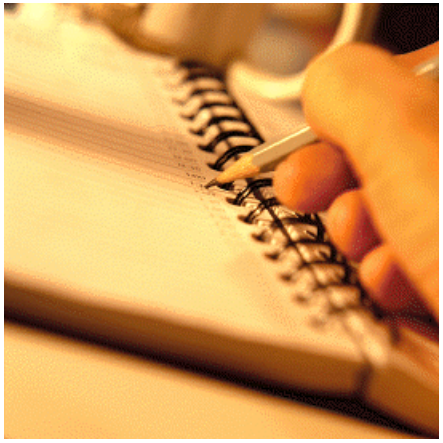
2013-2014

Avec des éléments de cours de Richard Grin et Sébastien Mosser

http://miageprojet2.unice.fr/User:PhilippeCollet/Projet_de_d%C3%A9veloppement_2013-2014

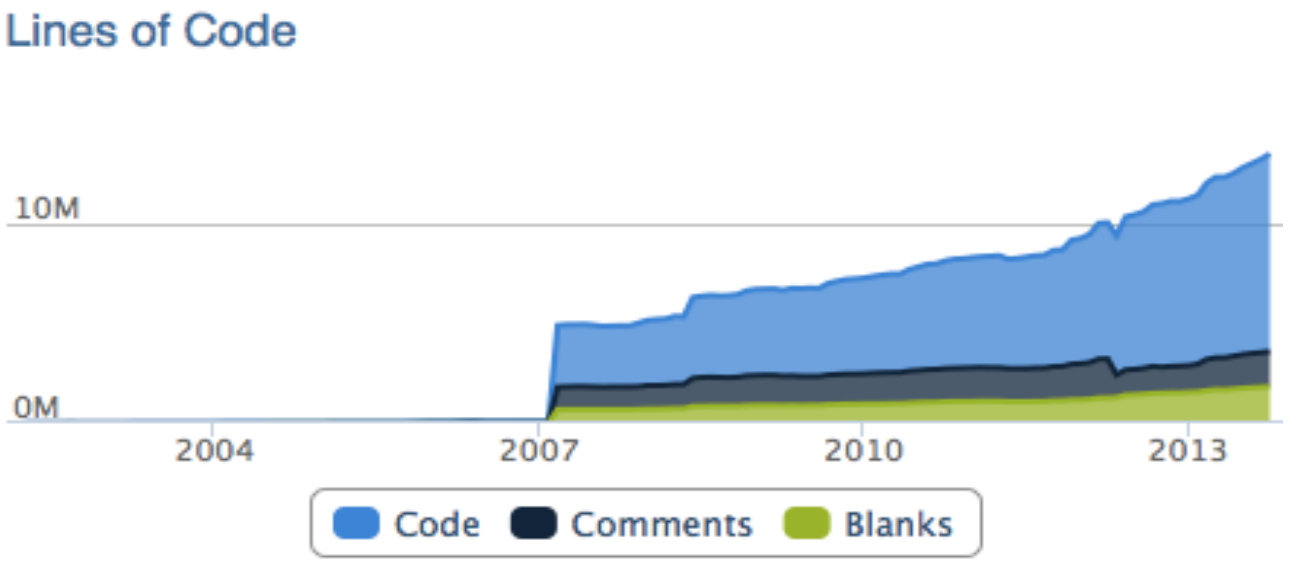
Université Nice SOPHIA ANTIPOLIS

Plan



- Ant
- Maven

- Exemple de Firefox (plus de 9M LOC)



- 1 commit tous les 14 minutes

- Chaque commit nécessite 137 heures de calcul

http://www.ohloh.net/p/firefox

http://oduinn.com/blog/2012/08/21/137-hours-compute-hours-every-6-minutes/

Problèmes

- Ce n'est pas gérable « à la main »

- Ce n'est pas gérable dans votre IDE favori
  - Le code est sur un serveur, etc.

- **Mais ce sont juste des dépendances entre modules, classes, fichiers, trucs…**

Construction automatique en Java
1ère étape : ANT

- http://ant.apache.org/

Université Nice SOPHIA ANTIPOLIS

Introduction

- Syntaxe et options très fournies
- Projet Open source (fondation Apache)
- Ecrit lui-même en Java

Principes

- Modèle de la commande make
  - un projet
  - des cibles (compile, jar, javadoc,…)
  - La description des cibles et les dépendances entre les cibles sont décrites dans un fichier
  - *Fichier XML, nommé par défaut build.xml*
- Extensible : on peut ajouter ses propres commandes/tâches

build.xml : exemple

```
<project name="hello" default="compile">


  <target name="prepare">
    <mkdir dir="./classes" />
  </target>


  <target name="compile" depends="prepare">
    <javac srcdir="./src"
           destdir="./classes" />
  </target>
</project>
```

Script de construction : structure

- une en-tête XML (avec l'indication optionnelle d'une DTD)

- une entrée `project` qui contient
  - optionnellement, des entrées `property`
  - optionnellement, des entrées path ou classpath
  - une ou plusieurs entrées `target`
  - optionnellement, une entrée `description`
    - Description informelle du projet
    -
```
<description>
Ce projet permet de . . .
. . .
</description>
```

Entrée project

- Chaque fichier de construction contient une et une seule entrée project

- Cette entrée peut avoir 3 attributs
  - `name` le nom du projet
  - `default` la cible par défaut (requis)
  - `basedir` le répertoire de base pour les chemins relatifs
    - peut être écrasé par la propriété `basedir`
    - par défaut le répertoire où se trouve le fichier de construction

Les cibles

- Une cible (`target`)
  - correspond à une action qui est décrite dans le fichier
  - peut dépendre d'autres cibles (attribut `depends`)
- Chaque type de cible peut avoir ses propres attributs
- Les attributs communs à toutes les cibles :
  - `name` : le nom de la cible (obligatoire)
  - `description` : si elle apparaît, permet de lister une description de la cible avec l'option `-projecthelp` de l'appel de `ant`
  - `depends` : permet d'indiquer les autres cibles dont dépend une cible

Dépendances de cibles

- On peut indiquer plusieurs cibles dont une cible dépend (`depends A,B,C` par exemple)
  - les cibles seront exécutées dans l'ordre du `depends` (de gauche à droite)

- Dans la gestion des dépendances, les tâches ne sont exécutées qu'une fois :

```
    <target name="A"/>
<target name="B" depends="A"/>
<target name="C" depends="A,B"/>
```

A ne sera exécuté qu'une seule fois

Comportement sur erreur

- Le plus souvent, une erreur dans une tâche arrête la construction de la cible correspondante
  - Une classe ne compile pas, la cible qui construit le jar s'arrête

- Certaines tâches ne provoque pas d'arrêt
  - On peut leur ajouter un attribut « `failonerror` » à `true` pour forcer l'arrêt
  - Exemple : la tâche « `java` »

Tâches

- Une tâche est une unité d'exécution « élémentaire » pour réaliser une cible

- Attributs possibles :
  - id donne un identificateur unique à la tâche ; cet identificateur peut être utilisé dans le reste du fichier pour désigner la tâche
  - taskname donne un autre nom à la tâche ; ce nom sera utilisé dans les rapports d'exécution
  - description décrit la tâche (texte non formaté)

- Les tâches optionnelles
  - nécessitent une bibliothèque supplémentaire pour être exécutées (fichier .jar à installer)

Tâches (java)

- Ant fournit des tags XML pour les tâches les plus communes en Java :
  - javac, java, rmic, javadoc, jar, unjar, war, unwar, ear

```
<javadoc
  packagenames=« com.bigmoney.pack.*"
  sourcepath="${src}"
  destdir="${doc}/api"
  use="true"/>
```

La tâche javac

- Compilateur utilisé : propriété `build.compiler`
  - par défaut, JDK qui exécute Ant
- Compiler récursivement tous les fichiers java du répertoire des sources
  - Utilisation des dates de dernière modification pour savoir si une classe a besoin d'être recompilée
- Très grand nombre d'attributs : `srcdir` (requis), `classpath`, `debug`, `optimize`, `source`, `fork`…

```
<javac srcdir="${src}“ destdir="${build}"
  classpath="xyz.jar" debug="on" />
```

La tâche java

- Lance l'exécution d'un programme java
- Attributs :
  - `classname` ou `jar` pour indiquer la classe à exécuter
  - `classpath`, `fork`, `failonerror`, `output`, `append`…

```
<java jar="dist/test.jar" fork="true"
    failonerror="true" maxmemory="128m">
<arg value="-h"/>
<classpath>
  <pathelement location="dist/test.jar"/>
  <pathelement path="${java.class.path}"/>
</classpath>

</java>
```

Exécution de Ant

- « `ant` » lance Ant en utilisant
  - le fichier `build.xml` du répertoire courant
  - la cible par défaut
    - on peut donner une autre cible en argument

- Options
  - `-buildfile` pour utiliser un autre fichier que `build.xml`
  - `-D`*`propriété=valeur`* pour donner la valeur d'une propriété
  - `-help` affiche les options disponibles
  - `-projecthelp` affiche une description du projet, avec toutes les cibles (*targets*) qui ont une description

Classpath (ou path)

❒ Permet d'indiquer le *classpath* :

```
<classpath>
  <pathelement path="${classpath}"/>
  <pathelement location="lib/helper.jar"/> </
classpath>
```

peut contenir plusieurs entrées

ne peut contenir qu'une entrée

❒ Les éléments sont indiqués par des entrées `pathelement` ou `fileset`

Comment faire mieux ?

❒ Regrouper des outils open source sous un chapeau commun pour gérer des projets, par exemple :

- Ant pour la construction
- JUnit pour le test unitaire (cf. cours suivants…)
- Jalopy pour formater le code source
- Checkstyle pour valider le code Java envers des standards de codage
- Javadoc pour la doc Java

❒ Gère des tâches comme des rapports, des dépendances, des configurations, des releases, des distributions, etc.

Construction automatique en Java
2ème étape : Maven

- http://maven.apache.org/

Université Nice SOPHIA ANTIPOLIS

Maven : principes

❒ Création d'un projet

```
mvn archetype:create -DgroupId=com.mycompany.app -DartifactId=my-app
```

❒ Structure par défaut :

```
my-app
|-- pom.xml
`-- src
    |-- main
    |   `-- java
    |       `-- com
    |           `-- mycompany
    |               `-- app
    |                   `-- App.java
    `-- test
        `-- java
            `-- com
                `-- mycompany
                    `-- app
                        `-- AppTest.java
```

Maven : pom.xml

- Le fichier central de toute configuration
  - Contient la majorité des informations sur le projet
  - Devient très long et très complexe (généré et modifié par interfaces graphiques)

```
<project xmlns="http://maven.apache.org/POM/4.0.0" ... >
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 ...">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Maven Quick Start Archetype</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>
```

Autre principes

- Description du projet : fichier POM (XML)
  - Identifier (groupId)
  - Target
  - Description
  - Stakeholders (développeurs, organisation)

Maven : phases

```
mvn archetype:create -DgroupId=com.mycompany.app -DartifactId=my-app
```

- archetype:create
  - archetype : nom du plugin => organisation en plugin avec dépendances
  - create : but (goal), similaire aux tâches ant

```
$ mvn package
...
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 2 seconds
[INFO] Finished at: Thu Oct 05 21:16:04 CDT 2006
[INFO] Final Memory: 3M/6M
[INFO] ------------------------------------------------------------------------
```

- package
  - Est une phase : une étape du cycle de construction
  - Le cycle de construction est une suite ordonnée de phases
  - L'exécution de la phase exécute toute les phases précédentes dans l'ordre

Maven : phases (suite)

```
mvn compile
```

- Exécute les phases suivantes
  1. validate
  2. generate-sources
  3. process-sources
  4. generate-resources
  5. process-resources
  6. compile
- Phases par défaut :
  - validate: validation du projet et de toutes les informations nécessaires (dépendances)
  - compile: compilation du code source
  - test: test du source compilé avec un framework de test (déclaré). Ces tests ne doivent pas nécessiter que le code soit packagé ou déployé
  - package: création d'un package distribuable (Jar par ex.) à partir du code compilé
  - integration-test: déploiement le package si nécessaire dans un environnement où des tests d'intégration sont exécutés
  - verify: exécution de vérifications sur la validité du package ou des critères de qualité
  - install: installation du package dans le repository local, pour être utilisable en dépendances d'autres projets locaux
  - deploy: copie du package dans un repository distant pour partage
- Autres phases très utiles :
  - clean: nettoyage !
  - site: génération du site web de documentation du projet

Exploiter le fichier POM

- mvn site
  - Génère le site web du projet

- mvn install
  - Compile et installe le projet…

Dépendences

```
<project>
[...]
<dependencies>
 <dependency>
   <groupId>log4j</groupId>
   <artifactId>log4j</artifactId>
   <version>1.2.14</version>
 </dependency>
 <dependency>
   <groupId>junit</groupId>
   <artifactId>junit</artifactId>
   <version>3.8.1</version>
   <scope>test</scope>
 </dependency>
</dependencies>
[...]
</project>
```

- Télécharge les jars (automatiquement !)
- Setup du classpath (pour les tests avec junit)
- Les dépendances sont tenues à jour !

Architecture du Repository