TP AL ESIR2

Introduction à Maven et à l'intégration continue[1]

Objectifs du TP

Avant de travailler sur des TP de test logiciel, vous allez utiliser dans ce premier TP des outils liés au processus de développement logiciel, en l'occurrence Maven et des outils d'intégration continue souvent associés tels que Sonar et Jenkins. Le TP de test qui suivront utiliserons Maven. Le but de ce TP est donc de :

- **Comprendre le fonctionnement de Maven**
- **Utiliser les artefacts**
- **Configurer un projet IntelliJ avec Maven**
- **Créer son propre MOJO**
- **Générer des rapports Maven**
- **Utiliser git pour sauvegarder le code source de votre projet**
- **Utiliser des outils d'intégration continue**

Liens utiles

- Site de Maven : http://maven.apache.org/
- Plugin Checkstyle : http://maven.apache.org/plugins/maven-checkstyle-plugin/
- FAQ maven developpez.com : http://java.developpez.com/faq/maven/

Environnement

Selon le 3ième lien donné ci-dessus, Maven est essentiellement un outil de gestion et de compréhension de projet. Maven offre des fonctionnalités de :

- Construction, compilation
- Documentation
- Rapport
- Gestion des dépendances
- Gestion des sources
- Mise à jour de projet
- Déploiement

1Ce TP a été créé à l'origine par Olivier Barais (olivier.barais@irisa.fr), puis amélioré par Arnaud Blouin (arnaud.blouin@irisa.fr)

Utiliser Maven consiste à définir dans chaque projet à gérer un script Maven appelés POM : *pom.xml*. Nous allons voir dans ce TP qu'un POM permet de définir des dépendances, des configurations pour notamment construire, tester, mettre en paquet des artefacts logiciels (exécutables, tests, documentations, archives, etc.). Pour cela, Maven récupère sur des dépôts maven les outils dont il a besoin pour exécuter le POM. Utiliser Maven requière donc : une (bonne) connexion à Internet car il télécharge beaucoup de choses ; de l'espace disque pour la même raison. Les artefacts qu'il télécharge sont originellement stockés dans le dossier *.m2* dans votre home-dir. Ce dossier contient également le fichier de configuration Maven : *settings.xml*.
La première étape consiste donc à configurer Maven pour changer l'endroit où les artefacts téléchargés seront stockés afin d'éviter des problèmes d'espace disque. Pour ce faire, utilisez le fichier *settings.xml* suivant à mettre donc dans le dossier *.m2* :

```
<?xml version="1.0" encoding="UTF-8"?>
<settings>
    <localRepository>H:\mavenrepository</localRepository>
    <offline>false</offline>
</settings>
```

Partie 1 : Utilisation de maven

Pour initialiser un projet Java basique, vous pouvez utiliser l'archetype maven : *maven-archetype-quickstart*. Un archetype est une sorte de modèle dont le but est de faciliter la mise en place de projets. Vous avez juste à fournir un *groupid* et un *artefactid*. Un *groupid* est l'identifiant du groupe de projets dont le projet fait partie. Un *artefactid* est l'identifiant du projet au sein de son groupe.

```
mvn archetype:generate -DgroupId=[your project's group id] -DartifactId=[your project's artifact id] -DarchetypeArtifactId=[archetype artifact id]
```

Vous obtenez alors la structure de projet suivante :

```
.
└── [artifact id]
    ├── pom.xml
    └── src
        ├── main
        │   └── java
        │       └── [your group id]
        │           └── App.java
        └── test
            └── java
                └── [your group id]
                    └── AppTest.java
```

Par exemple si vous exécutez la commande :

```
mvn archetype:generate -DgroupId=fr.esir -DartifactId=tpmaven
-DarchetypeArtifactId=maven-archetype-quickstart
```

Vous obtiendrez l'arborescence suivante.

```
tpmaven
├── pom.xml
└── src
    ├── main
    │   └── java
    │       └── fr
    │           └── esir
```

```
        └── App.java
└── test
    └── java
        └── fr
            └── esir
                └── AppTest.java
```

Partie 2 : Configuration d'IntelliJ

Tous les IDE modernes (Eclipse, IntelliJ…) supportent très bien maven. Dans ce TP nous allons utiliser IntelliJ, toute fois les explications pour Eclipse sont facilement trouvable en ligne (par exemple http://www.vogella.com/tutorials/EclipseMaven/article.html).

Afin d'importer le projet que vous avez créé en ligne de commande, ouvrez IntelliJ, puis faites « File » et «Open... ». Choisissez le répertoire contenant le projet maven créé dans la Partie 1, et validez.

IntelliJ possède nativement une façon d'appeler Maven facilement. Pour cela, faites « View », « Tool Windows », « Maven Projects ». Un panneau s'affiche à droite avec la liste des projets maven détectés, et l'ensemble des actions Maven qu'il est possible de faire sur le projet. Effectuer ces actions en double cliquant dans ce panneau est équivalent à taper en ligne de commande « mvn <action> ».

De plus, il est important de s'assurer que votre projet s'exécutera bien quelque soit la machine sur laquelle il sera exécuté. Deux problèmes récurrents sont la version de java utilisée pour la compilation et l'exécution, ainsi que le charset du code source.

Afin d'anticiper ces problèmes nous allons définir explicitement la version de la jvm et le charset dans le projet maven. Pour cela il faut ouvrir le fichier pom.xml situé à la racine de votre projet, et y définir les propriétés suivantes :

```
<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">

    <!-- ... -->
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
    </properties>
    <!-- ... -->
</project>
```

Une fois ces configurations ajoutées, il faut demander à IntelliJ de les prendre en compte en faisait : clic-droit sur le projet → Maven → Reimport.

Vous pouvez vérifier que les configurations sont bien prises en compte en modifiant temporairement la version 1.8 par 1.7, rééxécuter le réimport, puis constater dans les configurations du projet (clic droit → Open Module Settings) que le Language level est bien cohérent avec la configuration maven.

En cas de problème de localisation du JDK

Pour configurer le JDK dans eclipse: Windows -> Preferences -> Java -> Installed JREs -> Search button. Vous trouverez un jdk 1.8 dans C:\Programs\Java

Pour configurer le JDK dans IntelliJ : File → Project Structure → SDKs → Cliquer sur le plus vert → JDK → Vous trouverez un jdk 1.8 dans C:\Programs\Java

Partie 3 : Génération de rapports

Générer la javadoc

Ajoutez des commentaires dans le code de votre projet. Ajoutez le code suivant dans le *pom.xml* de votre projet.

```
<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">
    <!-- ... -->
    <reporting>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-javadoc-plugin</artifactId>
                <version>3.0.0</version>
            </plugin>
        </plugins>
    </reporting>
</project>
```

Puis lancez : *mvn site*. Ce goal crée un site Web pour votre projet. Par défaut, les goals maven générant des fichiers travaillent dans le dossier *target* se trouvant au même niveau que le fichier *pom.xml*. Allez dans le dossier *target/site* et ouvrez le fichier *index.html*. Vous pouvez regarder la Javadoc générée en cliquant sur *Project reports*.

Valider la qualité du code avec le plugin *checkstyle*

Ajoutez à la section <plugins> dans <reporting> le plugin *checkstyle* :

```
<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">
    <!-- ... -->
    <reporting>
        <plugins>
            <!-- ... -->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-checkstyle-plugin</artifactId>
                <version>3.0.0</version>
            </plugin>
            <!-- ... -->
        </plugins>
    </reporting>
</project>
```

Lancez *mvn clean site* (le goal *clean* vide le dossier *target*). Une nouvelle section *Checkstyle* a été ajouté dans *Project reports*. Quelle est la norme de codage à laquelle se réfère le rapport par défaut ? Modifier le pom pour utiliser la norme de codage maven. Liens utiles (notamment le second) pour répondre à ces questions :

Site de de l'outil CheckStyle : http://checkstyle.sourceforge.net/

Site du plugin Maven : http://maven.apache.org/plugins/maven-checkstyle-plugin/

Rapport croisé de source

```
<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">
    <!-- ... -->
    <reporting>
        <plugins>
            <!-- ... -->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-jxr-plugin</artifactId>
                <version>2.5</version>
            </plugin>
            <!-- ... -->
        </plugins>
    </reporting>
</project>
```

Lien utile : http://maven.apache.org/plugins/maven-jxr-plugin/. Quelle est la valeur ajoutée de ce plugin ? En particulier, montrez sa complémentarité avec *CheckStyle*. Désormais vous pouvez passer du rapport *CheckStyle* au code source en cliquant sur le numéro de ligne associé au commentaire *CheckStyle*.

Connaître l'activité du projet

But : avoir des informations sur les fichiers modifiés et leurs auteurs. Pour cela vous allez versionner en local votre projet avec *git*. Aller dans le dossier de votre projet, puis :

- créez un dépôt git : *git init* ;
- le dossier *target* ne doit pas être commité, de même que les fichiers créés par IntelliJ : créez un fichier « .gitignore », et mettez y :

```
target
*.iml
.idea
```

- versionnez tous les autres fichiers : *git add --all* . Notez qu'on se permet de faire ça uniquement parce qu'on a bien pris soin de créer un *.gitignore* qui empêche de versionner les fichiers inutiles ! Plus généralement, on utilise très peu le *–all*, à part éventuellement pour un premier commit, comme ici.

- Vérifier la liste des fichiers en attente d'être commit *git status*
- commitez : *git commit -m "first commit"*.

Ajoutez ensuite dans la section <plugins> de <reporting> du pom le plugin *changelog* :

```
<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">
    <!-- ... -->
    <reporting>
        <plugins>
            <!-- ... -->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-changelog-plugin</artifactId>
                <version>2.3</version>
            </plugin>
            <!-- ... -->
        </plugins>
    </reporting>
</project>
```

Ajoutez dans le bloc <project> le lien vers votre projet :

```
<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">
    <!-- ... -->
    <scm>
        <connection>scm:git:file://${project.basedir}</connection>
    </scm>
    <!-- ... -->
</project>
```

`${project.basedir}` va être automatiquement remplacé par le chemin absolu vers la racine du projet. Ainsi, la portabilité de votre projet peut être récupéré sur n'importe quelle machine (par exemple celle de votre bînome) et fonctionner sans avoir à intervenir sur les configurations maven.

Pour observer l'impact de `${project.basedir}` sur votre projet, vous pouvez faire clic droit → maven → effective pom et constater que la valeur de scm/connection est bien substituée par le chemin absolu vers votre projet.

Lancez *mvn clean site*. Le dossier */target/site* contient maintenant trois rapports d'activité :

- *changelog* : rapport indiquant toutes les activités sur le SCM.
- *dev-activity* : rapport indiquant par développeur le nombre de commits, de fichiers modifiés. Pour que ce rapport fonctionne, il faut déclarer les développeurs du projet dans le bloc <project>, de la sorte :

```
      <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">
    <!-- ... -->
    <developers>
        <developer>
            <name>Firstname Lastname</name>
            <email>email@address.org</email>
            <roles>
                <role>admin</role>
                <role>developer</role>
            </roles>
        </developer>
    </developers>
    <!-- ... -->
</project>
```

- *file-activity* : rapport indiquant les fichiers qui ont été révisés.

Partie 4 : Intégration Continue

Vous allez étudier les outils d'intégration continue Sonar et Jenkins. La différence entre ces deux outils est simple : Sonar est un outil d'assurance qualité tandis que Jenkins est un outils de « release engineering ». Les deux sont évidemment complémentaire.

Intégration avec l'outil Sonar

Téléchargez Sonar : https://sonarsource.bintray.com/Distribution/sonarqube/sonarqube-7.0.zip

De-compressez-le dans */tmp* (ou autre répertoire temporaire – attention à ne pas avoir le moindre espace ou caractère spécial dans le chemin complet du répertoire extrait !) et lancez-le :wget

```
(linux) sh /tmp/sonarqube-7.0/bin/linux-x86-64/sonar.sh start
(windows) bin\windows-x86-64\StartSonar.bat
```

Les explications d'exécution de sonar à l'aide de maven sont aussi expliquée, après connexion avec les identifiants admin/admin sur votre instance locale de sonarqube

Dans votre projet, lancez *mvn sonar:sonar.* Cela va faire appel à un plugin maven pour sonar, qui va contacter le serveur sonar lancé (par défaut, il va aller voir en local sur http://localhost:9000/) et lui donner les informations sur le projet.

Allez à l'adresse http://localhost:9000/. Loguez vous avec le login *admin* et le mot-de-passe *admin*. Allez dans *Quality Profiles* et changez les règles de qualités utilisées puis relancez *mvn sonar:sonar*. Baladez vous dans Sonar pour explorer ces différentes fonctionnalités. Vous pourrez ensuite arrêter sonar avec :

```
(linux) sh /tmp/sonarqube-4.0/bin/linux-x86-64/sonar.sh stop
(windows) ???
```

Intégration avec Jenkins

Sur http://jenkins-ci.org/, prenez la version **LTS** Java Web Archive (.war) pour la mettre dans */tmp* ou autre répertoire temporaire (http://mirrors.jenkins.io/war-stable/latest/jenkins.war).

Vous pouvez également déplacer l'endroit où la configuration Jenkins sera stockée, afin de ne pas « polluer » votre homedir :

(linux) export JENKINS_HOME=/tmp/.jenkins

(windows) ???

Démarrez jenkins : *java -jar jenkins.war.* Allez dans votre navigateur : http://localhost:8080/.

Pour créer un job, vous aurez besoin d'indiquer à Jenkins où se trouve votre *repository* git. Il y a deux façons de procéder :

(a) Étant donné que git est un système décentralité, votre répertoire de projet un *repository* git fonctionnel, même s'il est seulement accessible localement. Vous pouvez donc indiquer à Jenkins un chemin local vers votre répertoire de projet.

(b) Ceci étant, dans le cadre d'un développement collaboratif, on utilise souvent un repository « central » par convention (sur un serveur géré par le groupe de développement, ou bien sur github, etc.). Dans ce cas, on indiquera à Jenkins d'utiliser ce repo principal pour l'intégration continue. Vous pouvez donc si vous le souhaitez mettre votre code sur github de cette manière :

- créez un nouveau *repository* via l'interface github
- liez votre dépôt local au distant : *git remote add origin git@github.com:login/nomRepo.git*
- mettez votre code sur ce dépôt : *git push origin master*

De retour dans Jenkins créez un job en cliquant sur New item. Donnez lui un nom et sélectionnez le type Pipeline avant de cliquer sur OK.

Dans l'onglet Pipeline, sélectionner *Pipeline script from SCM* dans la liste déroulante.

Choisir le SCM git et saisir l'adresse de votre dépôt git dans le champ *Repository URL* soit (a) en indiquant un chemin local (e.g. *file:///home/toto/monProjet*), soit (b) en indiquant l'url du repository git que vous avez préalablement créé sur github (i.e. *https://github.com/login/nomRepo.git*).

Vous avez maintenant configuré Jenkin. Malgré tout lancer un build (via *Build Now*) va pour l'instant provoquer une erreur *java.io.FileNotFoundException*

En effet, l'exécution du pipeline Jenkins s'attend à trouver un ficher *Jenkinsfile* à la racine de votre projet.

Ce fichier défini les étapes nécéssaires à la construction de votre projet.

Créer ce fichier à la racine de votre projet et poussez le sur votre repository, avec le contenu suivant :

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                sh 'mvn -B -DskipTests clean package'
            }
        }
        stage('Test') {
            steps {
                sh 'mvn test'
            }
            post {
                always {
                    junit 'target/surefire-reports/*.xml'
                }
            }
        }

        stage('Archive') {
```

```
            steps {
                archiveArtifacts(artifacts: 'target/')
            }
        }
    }
}
```

Vous pouvez maintenant relancer votre job jenkins et voir apparaître 3 stages sur la page du job : *Checkout SCM*, *Build*, *Test* et *Archive* (tous sur fond vert si le job est réussi, en rouge sinon).

L'intégration de sonar se fait en deux étapes.

Première il faut ajouter un stage dans le pipeline :

```
stage('SonarQube analysis') {
    steps {
        sh "mvn org.sonarsource.scanner.maven:sonar-maven-plugin:3.2:sonar
-Dsonar.host.url=\"${SONAR_HOST_URL}\" -Dsonar.login=\"${SONAR_LOGIN}\""
    }
}
```

Ce stage contient deux paramètres : SONAR_HOST_URL et SONAR_LOGIN, que nous ne souhaitons par écrire en dur dans le Jenkins file pour des raisons de sécurité (ne jamais versionner de clés de sécurité) et de portabilité (ne jamais versionner de valeurs configuration comme des adresses ip).

Pour définir les valeurs de ces deux paramètres, nous devons rendre le build paramètrable.

Dans les options de configurations du job jenkins, cochez *this project is parametrized* et ajouter deux *string parameters* avec :

Name : SONAR_HOST_URL ; Default Value : http://localhost:9000

Name : SONAR_LOGIN ; Default Value : la clé générée lors de l'initialisation de sonar.

Une fois cela fait vous pouvez relancer le job via le bouton *Build with Parameters*

Etat des builds :

Dans l'historique des builds, un icône bleu doit apparaître à la fin de la construction pour désigner la construction correcte de l'artefact (bleu car le développeur de Jenkins est Japonais et au Japon le bleu équivaut au vert chez nous, d'ailleurs un plugin Jenkins existe pour afficher des icône verte et non bleue...). Cliquez ensuite sur le lien sous « Construction du module », les artefacts créés par jenkins en utilisant le POM du projet sont visibles dont un jar. Ouvrez ce dernier, vous verrez que le *manifest* est vide. Dans les étapes suivantes vous allez compléter le POM pour obtenir un vrai jar exécutable.

Partie 5 : Packager des artefacts logiciels avec maven

Comme expliqué précédemment, ces artefacts logiciels peuvent être produits soit en utilisant directement maven en ligne de commande, soit en utilisant Jenkins. Nous allons dans cette dernière partie étudier différents plugins maven permet de réaliser de nombreuses actions de liées à la construction d'artefacts logiciels. Notez que par défaut, maven produit un artefact très simple, à savoir un jar contenant toutes les classes compilées. Nous allons voir dans cette partie comment personnaliser à la fois la compilation, et la création des artefacts.

Création d'un jar exécutable via maven

Le plugin *maven-compiler-plugin* est chargé par défaut par maven, avec n'importe pom.xml, afin d'avoir directement accès à la possiblité pour maven de compiler le projet (goal *compile*). Ceci étant, il est possible de le déclarer explicitement dans la liste des plugins du projet afin de le configurer.

Ces plugins ne sont pas liés à la documentation, donc au lieu de travailler dans la section <reporting>, vous allez ajouter un bloc <build> dans le bloc <project> de votre POM. Dans ce nouveau bloc vont être ajouté les plugins utilisés/configurés dans cette partie comme le suivant :

```
<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">
    <!-- ... -->
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.7.0/version>
                <configuration>
                    <source>${maven.compiler.source}</source>
                    <target>${maven.compiler.target}</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
    <!-- ... -->
</project>
```

Comme vous pouvez le constater, on a ici ajouté un bloc <configuration> pour indiquer au plugin de compiler du code et du binaire compatibles Java 1.7. La documentation expliquer de nombreuses autres options de configuration pour la compilation.

Cf. : http://maven.apache.org/plugins/maven-compiler-plugin/

C’est une autre méthode pour définir la version de java à utiliser pour la compilation en plus de celle utilisée en début de TP.

Le plugin *maven-jar-plugin* est lui aussi chargé par défaut par maven, afin cette fois d'avoir directement accès à la possiblité pour maven de packager le projet (goal *package*, qui lancera par lui même le goal *compile* au préalable). Il est là aussi possible de le déclarer explicitement dans la liste des plugins du projet afin de le configurer.

Cette fois-ci, on ne vous donne pas le bloc xml à ajouter : trouvez comment déclarer ce plugin dans votre POM, et comment le configurer afin de créer automatiquement un manifest dans le jar dans lequel est indiqué quelle est la classe contenant la méthode *main* à utiliser. Vous aurez besoin pour cela de configurer le paramètre « archive ».

http://maven.apache.org/plugins/maven-jar-plugin/

Lancez *mvn clean install* et exécutez le nouveau jar généré.

Exécution de test via maven

Le plugin *maven-surefire-plugin* est lui aussi chargé par défaut par maven. Il a pour rôle d'exécuter les tests du projet à chaque fois que c'est demandé (goal test), mais également à chaque fois que le projet est

compilé (goals compile pour package. Ainsi, par défaut avec maven, la compilation échoue si des tests ne passent pas ! Rien à faire de particulier dans cette section, c'est simplement à titre informatif.

Création d'archives des sources et des exécutables

Le plugin *maven-assembly-plugin* permet de créer des archives. Ce plugin est notamment très utile pour créer des archives des sources ou des fichiers exécutables, cf :

http://maven.apache.org/plugins/maven-assembly-plugin/

Étudiez et adaptez l'utilisation de ce plugin dans le projet suivant :

https://github.com/arnobl/latexdraw/blob/master/latexdraw-core/net.sf.latexdraw/pom.xml

pour l'utiliser dans votre projet afin de créer un zip des sources et un autre contenant le jar exécutable.

Commitez les modifications sur github et relancez un build sur Jenkins afin d'observer les évolutions apportées.

Bonus : développement d'un plugin Maven

Il y a des outils qui sont bien pratiques, voire parfois nécessaires à tout développeur qui se respecte. Un logiciel de gestion de build comme Ant ou Maven en fait partie. Outre l'avantage indéniable de permettre de télécharger la moitié de l'internet lors du premier build sur une machine clean, Maven permet aussi via son système de plugins d'effectuer tout un tas d'actions allant plus loin que la compilation des classes et leur packaging dans un JAR – on en a vu un certain nombre dans le reste du TP. Nous allons ici voir comment développer un plugin pour Maven : un plugin qui compte le nombre de classes et d'attributs/méthodes par classe dans notre code.

Création du squelette de plugin Maven

Un plugin Maven n'est rien d'autre qu'un projet Maven avec un packaging de type *mavenplugin* :

Pour créer ce projet, nous utilisons l'artifact *mavenarchetypemojo*

```
mvn archetype:generate -DgroupId=fr.esir -DartifactId=classcounter
-DarchetypeArtifactId=maven-archetype-mojo
```

Nous obtenons le projet contenant le pom cijoint.

La convention veut que les plugins soient nommés xyz-maven-plugin, ce qui permet d'appeler les goals de la manière suivante lors d'un build :

```
$ mvn classcounter:some-goal
```

Notez qu'il n'est pas nécessaire de spécifier le nom complet du plugin, Maven va compléter avec "-maven-plugin", ça fait quelques caractères en moins à écrire ! Si la convention de nommage n'est pas suivie, il faut impérativement spécifier le nom complet du plugin dans la ligne de commande.

Ensuite, un plugin contient un ou plusieurs MOJO, qui vont exécuter le "vrai" code du plugin. Un MOJO correspond en fait à un goal. Par exemple, dans le plugin *dependency-maven-plugin*, on retrouve les goals "tree" (*mvn dependency:tree*) et "resolve" (*mvn dependency:resolve*)

Créons donc un squelette de MOJO pour notre générateur de code :

```
package fr.istic.master2.plugin.classcounter;
      import org.apache.maven.plugin.AbstractMojo;
```

```
import org.apache.maven.plugin.MojoExecutionException;
import org.apache.maven.plugin.MojoFailureException;

/**
 * A goal to generate code.
 *
 * @goal count
 * @phase compile
 */
public class ClassCounterMojo extends AbstractMojo {
 public void execute() throws MojoExecutionException,
MojoFailureException {
 getLog().info("Hello World!");
 }
}
```

Notre MOJO étend AbstractMojo, qui fournit toute l'infrastructure utile à la création d'un goal. La méthode abstraite execute() contient le code à exécuter, c'est donc à nous de l'implémenter. Pour l'instant elle ajoute une ligne de log saluant le monde (je sais, j'ai dit tout à l'heure qu'on ne se contenterait pas de faire un hello world...). La méthode *getLog()* est un exemple de facilités fournies par *l'AbstractMojo*. Notez également la Javadoc, qui contient deux tags particuliers. *@goal* spécifie le nom du goal correspondant à notre MOJO, ce tag est obligatoire. *@phase* permet de dire à Maven que notre plugin intervient durant la phase de génération des sources du lifecycle Maven (avant la compilation des sources, donc). Pour utiliser notre plugin depuis un autre projet, il faut d'abord l'installer dans notre dépôt local :

```
$ mvn install
```

Notre plugin est prêt à être exécuté. Configurons le pom.xml de notre projet tpmaven pour permettre l'exécution du plugin lors du build :

```
<build>
<plugins>
<plugin>
<groupId>fr.istic.master2</groupId>
<artifactId>classcounter-maven-plugin</artifactId>
<version>1.0-SNAPSHOT</version>
</plugin>
</plugins>
</build>
```

Puis lançons un build Maven :

```
$ mvn classcounter:count
[INFO] Scanning for projects...
[INFO]
[INFO] [INFO] Building classcountermavenplugin 1.0SNAPSHOT
[INFO]
[INFO]
```

```
[INFO] classcountermavenplugin:1.0SNAPSHOT:count (defaultcli) @
classcountermavenplugin
[INFO] Hello World!
[INFO]
[INFO] BUILD SUCCESS
[INFO]
[INFO] Total time: 0.322s
[INFO] Finished at: Mon May 14 14:47:54 CEST 2012
[INFO] Final Memory: 2M/81M
[INFO]
```

Ajout de paramètres au plugin. Notre cahier des charges requiert la possibilité de configurer certaines parties du générateur. Pour cela, ajoutons des paramètres à notre MOJO:

```
public class ClassCounterMojo extends AbstractMojo {
 /**
* Location of the file.
* @parameter expression="${project.build.directory}"
* @required
*/
private File outputDirectory;
/**
* Message language
* @parameter default-value="french"
* @required
*/
private String language;


 ......
}
```

Notre attribut utilise également des tags Javadoc pour indiquer que c'est un paramètre du plugin. Ce paramètre est obligatoire et prend une valeur par défaut.

L'attribut "alias" de @parameter permet de définir le nom du paramètre s'il est différent du nom de l'attribut Java. Ainsi, par défaut il y aura un paramètre nommé " language ", puisqu'on ne lui a pas donné d'alias. Une liste plus exhaustive des tags Javadoc est disponible sur le site de Sonatype. Ouvrons à nouveau le pom.xml de notre projet pour ajouter la configuration du plugin :

```
 <build>
 <plugins>
 <plugin> <groupId>fr.istic.master2</groupId>
 <artifactId>classcounter-maven-plugin</artifactId>
```

```
<version>1.0SNAPSHOT</version>
<configuration>
<language>english</language>
</configuration>
</plugin>
</plugins>
</build>
```

Pour vérifier que la configuration est bien injectée au runtime, ajoutons un log :

```
getLog().info("Indication will be given in the following language " +
language);
```

```
$ mvn classcounter:count
...
[INFO] classcountermavenplugin:1.0SNAPSHOT:count (defaultcli) @
classcountermavenplugin
[INFO] Hello World!
[INFO] Indication will be given in the following language English
[INFO]
[INFO] BUILD SUCCESS
...
```

Maintenant que nous savons créer, configurer et exécuter un plugin Maven, il ne reste plus qu'à implémenter la méthode execute() pour appeler les vrais services.

Meilleure gestion des paramètres

Comme vous l'avez vu, la configuration du Mojo se fait par défaut dans des tags Javadoc (des sortes de XDoclet). Ce n'est pas terrible dans la mesure où il est facile de faire une faute de frappe (ex @paramter au lieu de @parameter) qui ne sera pas détectée par le compilateur ni par Maven. Pour pallier ce problème, il existe un set d'annotations Java, le mavenpluginanno. Après avoir configuré la dépendance dans votre pom, il est ensuite possible de tout annoter :

```
<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">
    <!-- ... -->
    <dependencies>
        <!-- ... -->
        <dependency>
            <groupId>org.apache.maven.plugin-tools</groupId>
            <artifactId>maven-plugin-annotations</artifactId>
            <version>3.5.1</version>
            <optional>true</optional>
        </dependency>
        <!-- ... -->
```

```
    </dependencies>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-plugin-plugin</artifactId>
                <version>3.5.1</version>
                <executions>
                    <execution>
                        <id>default-descriptor</id>
                        <phase>process-classes</phase>
                    </execution>
                    <!-- if you want to generate help goal -->
                    <execution>
                        <id>help-goal</id>
                        <goals>
                            <goal>helpmojo</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

Et le code java :

```
package fr.esir;

// ...
import org.apache.maven.plugins.annotations.Mojo;
import org.apache.maven.plugins.annotations.Parameter;
// ...

@Mojo(name = "generate-annotated", defaultPhase =
LifecyclePhase.GENERATE_SOURCES)
public class AnnotatedMojo extends AbstractMojo {

    @Parameter(alias = "language", required = true)
    private File language;

    public void execute() throws MojoExecutionException {
        // ...
    }
}
```

Pour plus d'informations, il existe beaucoup de ressources en ligne, par exemple https://maven.apache.org/plugin-tools/maven-plugin-plugin/examples/using-annotations.html

Passons au travail de compter les classes, les attributs par classes et les opérations par classes. Pour cela nous allons utiliser la librairie clapper. Ajoutons clapper en dépendance à notre projet classcounter.

```
<dependency>
    <groupId>org.clapper</groupId>
```

```
    <artifactId>classutil_2.12</artifactId>
    <version>1.2.0</version>
</dependency>
```

Exemple de code pour la méthode execute du Mojo de notre plugin Maven.

```
package fr.esir;

// ...
import static scala.collection.JavaConverters.asJavaIterator;
import static scala.collection.JavaConverters.asScalaBuffer;
// ...

/**
 * Goal which touches a timestamp file.
 */
@Mojo(name = "touch", defaultPhase = LifecyclePhase.PROCESS_SOURCES)
public class AnnotatedMojo extends AbstractMojo {
    // …

    public void execute() throws MojoExecutionException {
        final File f1 = new File(outputDirectory.getAbsoluteFile(), "classes");
        final List<File> files = Arrays.asList(f1);
        final ClassFinder finder = new ClassFinder(asScalaBuffer(files));
        final Stream<ClassInfo> classes = finder.getClasses();
        final int classesCount = classes.size();
        if (Objects.equals("french", language)) {
            this.getLog().info("Nombre de classes " + classesCount);
        } else {
            this.getLog().info("Number of classes " + classesCount);
        }
        asJavaIterator(classes.iterator()).forEachRemaining(classInfo -> {
            final String name = classInfo.name();
            final int fieldsSize = classInfo.fields().size();
            final int methodsSize = classInfo.methods().size();
            if (Objects.equals("french", language)) {
                this.getLog().info("\t Pour la classe " + name);
                this.getLog().info("\t \t Nbre attributs " + fieldsSize);
                this.getLog().info("\t \t Nbre methodes " + methodsSize);
            } else {
                this.getLog().info("\t For the class named " + name);
                this.getLog().info("\t \t Number of filed " + fieldsSize);
                this.getLog().info("\t \t Number of methods " + methodsSize);
            }
        });
    }
}
```

Comme vous avez pu le voir, créer un plugin Maven n’est pas très compliqué. Les plugins sont suffisamment souples et configurables pour pouvoir exécuter du code déjà existant (par exemple remplacer un main() par un plugin Maven) sans trop de difficultés. La palette de plugins déjà disponibles est assez vaste, mais si un jour vous avez un besoin spécifique vous saurez que le coût de création d’un plugin n’est pas très élevé.

[1] http://labs.excilys.com/2012/05/14/monpremierpluginmaven/
[2] http://www.objis.com/formationjava/IMG/pdf/TP8_reporting.pdf