88. Des outils open source pour faciliter le développement

Niveau :  Supérieur

La communauté open source propose de nombreuses bibliothèques mais aussi des outils dont le but est de faciliter le travail des développeurs. Certains de ces outils sont détaillés dans des chapitres dédiés notamment Ant, Maven et JUnit. Ce chapitre va présenter d'autres outils open source pouvant être regroupés dans plusieurs catégories : contrôle de la qualité des sources et génération et mise en forme de code.

La génération de certains morceaux de code ou de fichiers de configuration peut parfois être fastidieuse voire même répétitive dans certains cas. Pour faciliter le travail des développeurs, des outils open source ont été développés par la communauté afin de générer ce code. Ce chapitre présente deux outils open source : XDoclet et Middlegen.

La qualité du code source est un facteur important pour tous développements. Ainsi certains outils permettent de faire des vérifications sur des règles de codification dans le code source. C'est le cas pour l'outil CheckStyle.

Ce chapitre contient plusieurs sections :

88.1. CheckStyle

CheckStyle est un outil open source qui propose de puissantes fonctionnalités pour appliquer des contrôles sur le respect de règles de codifications.

Pour définir les contrôles réalisés lors de son exécution CheckStyle utilise une configuration qui repose sur des modules. Cette configuration est définie dans un fichier XML qui précise les modules utilisés et pour chacun d'entre-eux leurs paramètres.

Le plus simple est d'utiliser le fichier de configuration nommé sun_checks.xml fourni avec CheckStyle. Cette configuration propose d'effectuer des contrôles du respect des normes de codification proposées par Sun. Il est aussi possible de définir son propre fichier de configuration.

Le site officiel de CheckStyle est à l'URL : http://checkstyle.sourceforge.net/

La version utilisée dans cette section est la 3.4

88.1.1. L'installation

Il faut télécharger le fichier checkstyle-3.4.zip sur le site de ChekStyle et le décompresser dans un répertoire du système.

La décompression de ce fichier crée un répertoire checkstyle-3.4 contenant lui-même les bibliothèques utiles et deux répertoires (docs et contrib).

CheckStyle peut s'utiliser de deux façons :

- en ligne de commande
- comme une tâche Ant ce qui permet d'automatiser son exécution

88.1.2. L'utilisation avec Ant

Pour utiliser CheckStyle, le plus simple est d'ajouter la bibliothèque checkstyle-all-3.4.jar au classpath.

Dans les exemples de cette section, la structure de répertoires suivante est utilisée :

/bin
/lib
/outils
/outils/lib
/outils/checkstyle
/src
/temp
/temp/checkstyle

Le fichier checkstyle-all-3.4.jar est copié dans le répertoire outils/lib et le fichier sun_checks.xml fourni par CheckStyle est copié dans le répertoire outils/checkstyle.

Il faut déclarer le tag CheckStyle dans Ant en utilisant le tag <taskdef> et définir une tâche qui va utiliser le tag <CheckStyle>.

Exemple : fichier source de test

```
1. public class MaClasse {
2.
3.   public static void main() {
4.     System.out.println("Bonjour");
5.   }
6.
7. }
```

Le fichier de build ci-dessous sera exécuté par Ant.

Exemple :

```
01. <project name="utilisation de checkstyle" default="compile" basedir=".">
02.   <!-- Definition des proprietes du projet -->
03.   <property name="projet.sources.dir"       value="src"/>
04.   <property name="projet.bin.dir"           value="bin"/>
05.   <property name="projet.lib.dir"           value="lib"/>
06.   <property name="projet.temp.dir"          value="temp"/>
07.   <property name="projet.outils.dir"        value="outils"/>
08.   <property name="projet.outils.lib.dir"    value="${projet.outils.dir}/lib"/>
09.
10.   <!-- Definition du classpath du projet -->
11.   <path id="projet.classpath">
12.     <fileset dir="${projet.lib.dir}">
13.       <include name="*.jar"/>
14.     </fileset>
15.     <fileset dir="${projet.outils.lib.dir}">
16.       <include name="*.jar"/>
17.     </fileset>
18.     <pathelement location="${projet.bin.dir}" />
19.   </path>
20.
21.   <!-- Declaration de la tache Ant permettant l'execution de checkstyle -->
22.   <taskdef resource="checkstyletask.properties"
23.     classpathref="projet.classpath" />
24.
25.   <!-- execution de checkstyle -->
26.   <target name="checkstyle" description="CheckStyle">
27.     <checkstyle config="outils/checkstyle/sun_checks.xml">
28.       <fileset dir="${projet.sources.dir}" includes="**/*.java"/>
29.       <formatter type="plain"/>
30.     </checkstyle>
31.   </target>
32.
33.   <!-- Compilation des classes du projet -->
34.   <target name="compile" depends="checkstyle" description="Compilation des classes">
35.     <javac srcdir="${projet.sources.dir}"
36.       destdir="${projet.bin.dir}"
37.       debug="on"
38.       optimize="off"
39.       deprecation="on">
40.       <classpath refid="projet.classpath"/>
41.     </javac>
42.   </target>
43.
44. </project>
```

Résultat :

```
C:\java\test\testcheckstyle>ant
Buildfile: build.xml
checkstyle:
[checkstyle] C:\java\test\testcheckstyle\src\package.html:0: Missing package doc
umentation file.
[checkstyle] C:\java\test\testcheckstyle\src\MaClasse.java:0: File does not end
with a newline.
[checkstyle] C:\java\test\testcheckstyle\src\MaClasse.java:2: Missing a Javadoc
comment.
[checkstyle] C:\java\test\testcheckstyle\src\MaClasse.java:2:1: Utility classes
should not have a public or default constructor.
[checkstyle] C:\java\test\testcheckstyle\src\MaClasse.java:4:3: Missing a Javado
c comment.
[checkstyle] C:\java\test\testcheckstyle\src\MaClasse.java:5: Line has trailing
spaces.
[checkstyle] C:\java\test\testcheckstyle\src\MaClasse.java:5:35: La ligne contie
nt un caractÞre tabulation.
[checkstyle] C:\java\test\testcheckstyle\src\MaClasse.java:7: Line has trailing
spaces.
[checkstyle] C:\java\test\testcheckstyle\src\MaClasse.java:7:1: La ligne contien
t un caractÞre tabulation.
BUILD FAILED
file:C:/java/test/testcheckstyle/build.xml:27: Got 9 errors.
       Total time: 7 seconds
```

Le tag <checkstyle> possède plusieurs attributs :

Nom	Rôle
file	précise le nom de l'unique fichier à vérifier. Pour préciser un ensemble de fichiers, il faut utiliser le tag fils <fileset>
config	précise le nom du fichier de configuration des modules de ChecksStyle
failOnViolation	précise si les traitements de l'outil Ant doivent être stoppés en cas d'échec des contrôles. La valeur par défaut est true
failureProperty	précise le nom d'une propriété qui sera valorisée en cas d'échec des contrôles

Exemple :

```
...
  <!-- execution de checkstyle -->
  <target name="checkstyle" description="CheckStyle">
    <checkstyle config="outils/checkstyle/sun_checks.xml" failOnViolation="false">
      <fileset dir="${projet.sources.dir}" includes="**/*.java"/>
      <formatter type="plain"/>
    </checkstyle>
  </target>
...
```

Résultat :

```
C:\java\test\testcheckstyle>ant
Buildfile: build.xml
checkstyle:
[checkstyle] C:\java\test\testcheckstyle\src\package.html:0: Missing package doc
umentation file.
[checkstyle] C:\java\test\testcheckstyle\src\MaClasse.java:2: Missing a Javadoc
comment.
[checkstyle] C:\java\test\testcheckstyle\src\MaClasse.java:2:1: Utility classes
should not have a public or default constructor.
[checkstyle] C:\java\test\testcheckstyle\src\MaClasse.java:4:3: Missing a Javado
c comment.
compile:
[javac] Compiling 1 source file to C:\java\test\testcheckstyle\bin
BUILD SUCCESSFUL
       Total time: 11 seconds
```

Il est possible de préciser deux types de formats de sortie des résultats lors de l'exécution de CheckStyle. Le format de sortie des résultats est précisé par un tag fils <formatter>. Ce tag possède deux attributs :

Nom	Rôle
type	précise le type. Deux valeurs sont possibles : plain (par défaut) et xml
toFile	précise un fichier dont le type déterminera le format des résultats stockés (par défaut la sortie standard de la console)

Exemple :

```
...
<!-- execution de checkstyle -->
<target name="checkstyle" description="CheckStyle">
```

```
  <checkstyle config="outils/checkstyle/sun_checks.xml" failOnViolation="false">
    <fileset dir="${projet.sources.dir}" includes="**/*.java"/>
    <formatter type="xml" toFile="${projet.temp.dir}/checkstyle_erreurs.xml"/>
  </checkstyle>
</target>
...
```

Il est alors possible d'appliquer une feuille de styles sur le fichier XML généré afin de créer un rapport dans un format dédié. L'exemple suivant utilise une feuille de style fournie par CheckStyle dans le répertoire contrib : cette feuille, nommée checkstyle-frames.xsl, est copiée dans le répertoire outils/checkstyle.

Exemple :

```
...
<!-- execution de checkstyle -->
<target name="checkstyle" description="CheckStyle">
  <checkstyle config="${projet.outils.dir}/checkstyle/sun_checks.xml" failOnViolation="false">
    <fileset dir="${projet.sources.dir}" includes="**/*.java"/>
    <formatter type="xml" toFile="${projet.temp.dir}/checkstyle/checkstyle_erreurs.xml"/>
  </checkstyle>
  <style in="${projet.temp.dir}/checkstyle/checkstyle_erreurs.xml"
    out="${projet.temp.dir}/checkstyle/checkstyle_rapport.htm"
    style="${projet.outils.dir}/checkstyle/checkstyle-frames.xsl"/>
</target>
...
```

Exemple :

```
C:\java\test\testcheckstyle>ant
Buildfile: build.xml
checkstyle:
[style] Processing C:\java\test\testcheckstyle\temp\checkstyle\checkstyle_er
reurs.xml to C:\java\test\testcheckstyle\temp\checkstyle\checkstyle_rapport.htm
[style] Loading stylesheet C:\java\test\testcheckstyle\outils\checkstyle\che
ckstyle-frames.xsl
compile:
BUILD SUCCESSFUL
Total time: 8 seconds
```

Il est possible d'utiliser d'autres feuilles de styles fournies par CheckStyle ou de définir la sienne.

88.1.3. L'utilisation en ligne de commandes

Pour utiliser CheckStyle en ligne de commandes, il faut ajouter le fichier checkstyle-all-3.4.jar au classpath par exemple en utilisant l'option -cp de l'interpréteur Java.

La classe à exécuter est com.puppycrawl.tools.checkstyle.Main

CheckStyle accepte plusieurs paramètres pour son exécution :

Option	Rôle
-c fichier_de_configuration	précise le fichier de configuration
-f format	précise le format (plain ou xml)
-o fichier	précise le fichier qui va contenir les résultats
-r	précise le répertoire dont les fichiers sources vont être récursivement traités

Exemple :

```
java -cp outils\lib\checkstyle-all-3.4.jar com.puppycrawl.tools.checkstyle.Main
  -c outils\checkstyle/sun_checks.xml -r src
```

Exemple :

```
...
C:\java\test\testcheckstyle>java -cp outils\lib\checkstyle-all-3.4.jar com.puppy
crawl.tools.checkstyle.Main -c outils\checkstyle/sun_checks.xml -r src
Starting audit...
C:\java\test\testcheckstyle\src\package.html:0: Missing package documentation fi
le.
src\MaClasse.java.bak:0: File does not end with a newline.
```

```
src\MaClasse.java:2: Missing a Javadoc comment.
src\MaClasse.java:2:1: Utility classes should not have a public or default const
ructor.
src\MaClasse.java:4:3: Missing a Javadoc comment.
Audit done.
...
```

88.2. Jalopy

Jalopy est un outil open source qui propose de formater les fichiers sources selon des règles définies.

Le site web officiel de Jalopy est http://jalopy.sourceforge.net

Jalopy propose entre autres les fonctionnalités suivantes :

- formatage des accolades selon plusieurs formats (C, Sun, GNU)
- indentation du code
- gestion des sauts de lignes
- génération automatique ou vérification des commentaires Javadoc
- ordonnancement des éléments qui composent la classe
- ajout de texte au début et à la fin de chaque fichier
- des plugins pour une intégration dans plusieurs outils : ant, Eclipse, Jbuilder, ...
- ...

Pour connaître les règles de formatage à appliquer, Jalopy utilise une convention qui est un ensemble de paramètres.

Jalopy peut être utilisé grâce à ses plugins de plusieurs façons notamment avec Ant, en ligne commande ou avec certains IDE.

La version de Jalopy utilisée dans cette section est la 1.0.B10

88.2.1. L'utilisation avec Ant

Le plus simple est d'utiliser Jalopy avec Ant pour automatiser son utilisation : pour cela, il faut télécharger le fichier jalopy-ant-0.6.2.zip et le décompresser dans un répertoire du système.

La structure de l'arborescence du projet utilisé dans cette section est la suivante :

/bin
/lib
/outils
/outils/lib
/src

Le répertoire src contient les sources Java à formater.

Les fichiers du répertoire lib de Jalopy sont copiés dans le répertoire outils/lib du projet.

Exemple : le fichier source qui sera formaté

```
public class MaClasse {public static void main() { System.out.println("Bonjour"); }}
```

Il faut définir un fichier build.xml pour Ant qui va contenir les différentes tâches du projet dont une permettant l'appel à Jalopy.

Exemple :

```
<project name="utilisation de jalopy" default="jalopy" basedir=".">
  <!-- Definition des proprietes du projet -->
  <property name="projet.sources.dir" value="src"/>
  <property name="projet.bin.dir" value="bin"/>
  <property name="projet.lib.dir" value="lib"/>
  <property name="projet.temp.dir" value="temp"/>
  <property name="projet.outils.dir" value="outils"/>
  <property name="projet.outils.lib.dir" value="${projet.outils.dir}/lib"/>

  <!-- Definition du classpath du projet -->
  <path id="projet.classpath">
    <fileset dir="${projet.lib.dir}">
      <include name="*.jar"/>
    </fileset>
    <fileset dir="${projet.outils.lib.dir}">
      <include name="*.jar"/>
    </fileset>
    <pathelement location="${projet.bin.dir}" />
```

```
  </path>

  <!-- Declaration de la tache Ant permettant l'execution de jalopy -->
  <taskdef name="jalopy"
         classname="de.hunsicker.jalopy.plugin.ant.AntPlugin"
      classpathref="projet.classpath" />

  <!-- execution de jalopy -->
  <target name="jalopy" description="Jalopy" depends="compile" >
    <jalopy loglevel="info"
            threads="2"
          classpathref="projet.classpath">
      <fileset dir="${projet.sources.dir}">
        <include name="**/*.java" />
      </fileset>
    </jalopy>
  </target>

  <!-- Compilation des classes du projet -->
  <target name="compile" description="Compilation des classes">
    <javac srcdir="${projet.sources.dir}"
           destdir="${projet.bin.dir}"
             debug="on"
          optimize="off"
        deprecation="on">
      <classpath refid="projet.classpath"/>
    </javac>
  </target>

</project>
```

Exemple :

```
C:\java\test\testjalopy>ant
Buildfile: build.xml

compile:
    [javac] Compiling 1 source file to C:\java\test\testjalopy\bin

jalopy:
   [jalopy] Jalopy Java Source Code Formatter 1.0b10
   [jalopy] Format 1 source file
   [jalopy] C:\java\test\testjalopy\src\MaClasse.java:0:0: Parse
   [jalopy] 1 source file formatted

BUILD SUCCESSFUL
Total time: 9 seconds
```

Suite à l'exécution de Jalopy, le code du fichier est reformaté.

Exemple :

```
public class MaClasse {
    public static void main() {
        System.out.println("Bonjour");
    }
}
```

Il est fortement recommandé de réaliser la tâche de compilation des sources avant leur formatage car pour assurer un formatage correct les sources doivent être correctes syntaxiquement parlant.

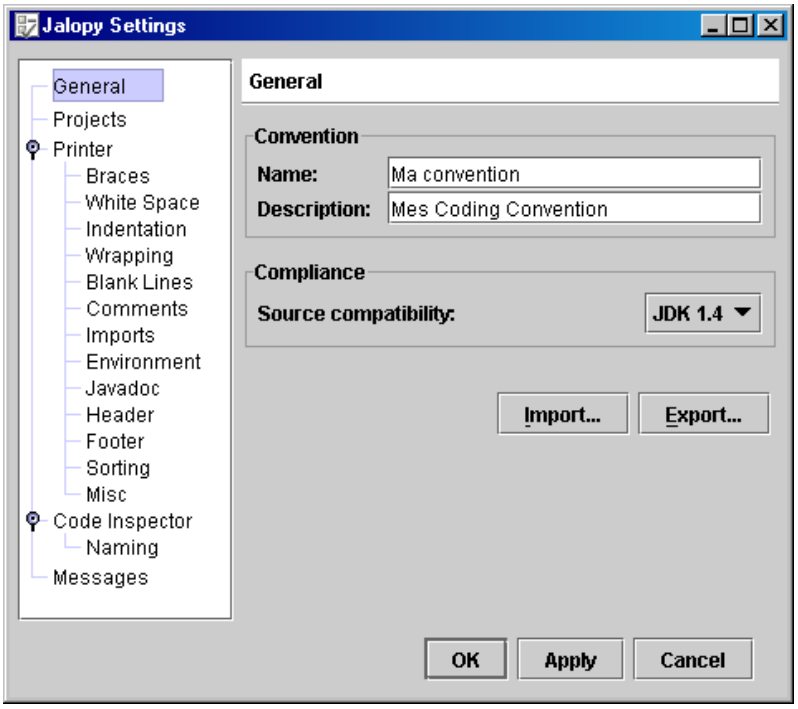
88.2.2. Les conventions

Jalopy est hautement paramétrable. Les options utilisées sont regroupées dans une convention.

Tous ses paramètres sont stockés dans le sous-répertoire .jalopy du répertoire Home de l'utilisateur.

Pour faciliter la gestion de ces paramètres, Jalopy propose un outil graphique qui permet de gérer les conventions.

Pour exécuter cet outil, il suffit de lancer le script preferences dans le répertoire bin de Jalopy (preferences.bat sous Windows et preferences.sh sous Unix).



Toutes les nombreuses options de formatage d'une convention peuvent être réglées par cet outil. Consultez la documentation fournie avec Jalopy pour avoir le détail de chaque option.

Une fonctionnalité particulièrement utile de cet outil est de proposer une prévisualisation d'un exemple mettant en oeuvre les options sélectionnées.

Voici un exemple avec quelques personnalisations notamment, une gestion des clauses import, la génération des commentaires Javadoc :

Exemple :

```
01. import java.util.*;
02.
03. public class MaClasse {
04.     public static void main() {
05.         List liste = new ArrayList();
06.         System.out.println("Bonjour");
07.     }
08.
09.     /**
10.      *
11.      */
12.     private int maMethode(int valeur) {
13.         return valeur * 2;
14.     }
15. }
```

Résultat :

```
01. C:\java\test\testjalopy>ant
02. Buildfile: build.xml
03.
04. compile:
05.     [javac] Compiling 1 source file to C:\java\test\testjalopy\bin
06.
07. jalopy:
08.    [jalopy] Jalopy Java Source Code Formatter 1.0b10
09.    [jalopy] Format 1 source file
10.    [jalopy] C:\java\test\testjalopy\src\MaClasse.java:0:0: Parse
11.    [jalopy] C:\java\test\testjalopy\src\MaClasse.java:1:0: On-demand import "jav
12. a.util.List" expanded
13.    [jalopy] C:\java\test\testjalopy\src\MaClasse.java:1:0: On-demand import "jav
14. a.util.ArrayList" expanded
15.    [jalopy] C:\java\test\testjalopy\src\MaClasse.java:11:1: Generated Javadoc co
16. mment
17.    [jalopy] C:\java\test\testjalopy\src\MaClasse.java:18:3: Generated Javadoc co
18. mment
19.    [jalopy] 1 source file formatted
20.
21. BUILD SUCCESSFUL
22. Total time: 10 seconds
```

Voici le source du code reformaté :

Exemple :

```java
//================================================================================
// fichier :       MaClasse.java
// projet :        $project$
//
// Modification :  date :       $Date$
//                 auteur :     $Author$
//                 revision :   $Revision$
//--------------------------------------------------------------------------------
// copyright:   JMD
//================================================================================

import java.util.ArrayList;
import java.util.List;


/**
 * DOCUMENT ME!
 *
 * @author $author$
 * @version $Revision$
 */
public class MaClasse {
  /**
   * DOCUMENT ME!
   */
  public static void main() {
    List liste = new ArrayList();
    System.out.println("Bonjour");
    System.out.println("");
  }

  /**
   * Calculer le double
   *
   * @param valeur DOCUMENT ME!
   *
   * @return DOCUMENT ME!
   */
  private int maMethode(int valeur) {
    return valeur * 2;
  }
}

// fin du fichier
```