COMPARATIF D'OUTILS D'ANALYSE DE CODE

OBJET :	Analyse d'outils
RÉFÉRENCE :	N/A
ÉQUIPE :	Ingénierie Méthode
AUTEUR(S) :	PERUCCA Jonathan
DESTINATAIRE(S) :	<Nom> <Prénom> – <Équipe>
VERSION / DATE :	1.0, le 11/01/2012
STATUT :	RELEASE CANDITATE
VALIDÉ PAR :	<Nom> <Prénom> – <Équipe>

Remarque

Sommaire

Cette documentation a pour but de présenter et d'analyser l'outils qui sera utilisé lors de notre projet afin de consolider au mieux les soucis de performance, d'optimisation et de sécurité du code.

Les technologies qui seront analysées seront Checkstyle, PMD et Findbugs pour lesquelles nous allons détailler leurs installations (par plugin eclipse et par Maven plugin), les principes de fonctionnement de chacun ainsi qu'un comparatif des outils sur lesquels nous allons nous pencher plus en détails.

Enfin, nous appuierons l'un de ces outils qui sera utilisé pour le projet UML Architect.

Checkstyle

Principe

CheckStyle permet de contrôler le respect des conventions de codage et d'avoir quelques métriques.

Ce contrôle est fait avec 126 règles et cela va du plus simple comme la longueur d'une ligne à des choses plus compliquées comme la complexité cyclomatique d'une classe.
Le plugin Maven met à disposition 4 fichiers de règles :

- config/sun_checks.xml

Convention Sun Microsystems (par défaut),
http://java.sun.com/docs/codeconv/

- config/maven_checks.xml

Convention de l'équipe Maven,
http://maven.apache.org/guides/development/guide-m2-development.html#maven%20code%20style

- config/turbine_checks.xml

Convention du projet Apache Turbine,
http://turbine.apache.org/common/code-standards.html

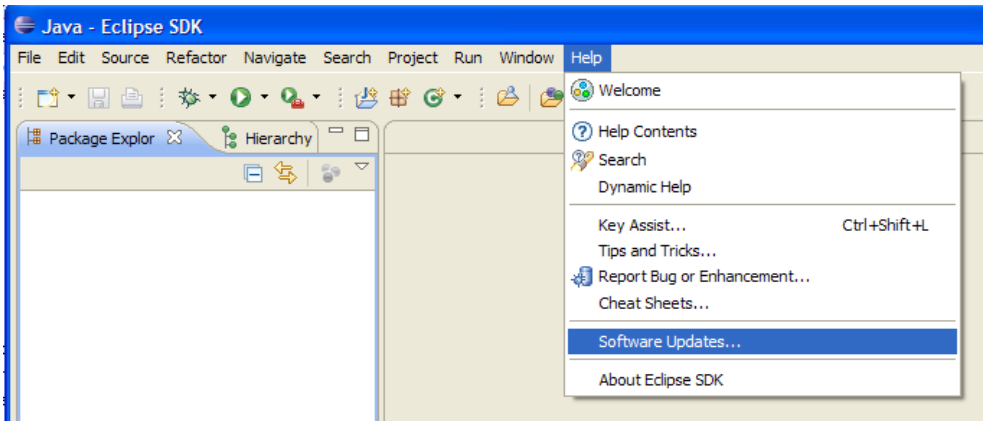
- config/avalon_checks.xml

Convention du projet Apache Avalon.
Ce projet n'existe plus.

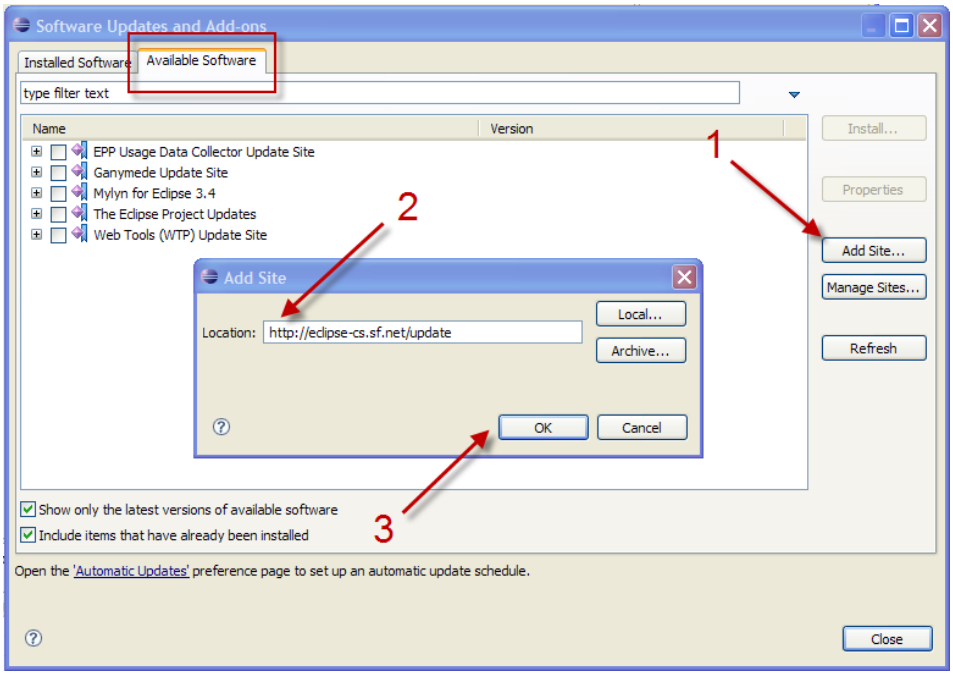
Procédure d'installation (plugin)

Avec Eclipse Installer

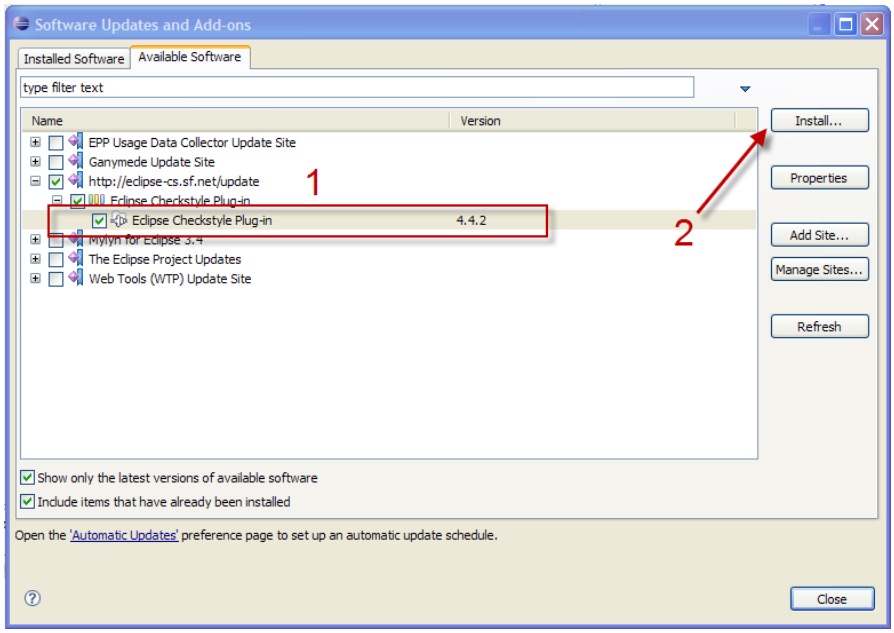
1. Aller dans Eclipse > Help > Software Update



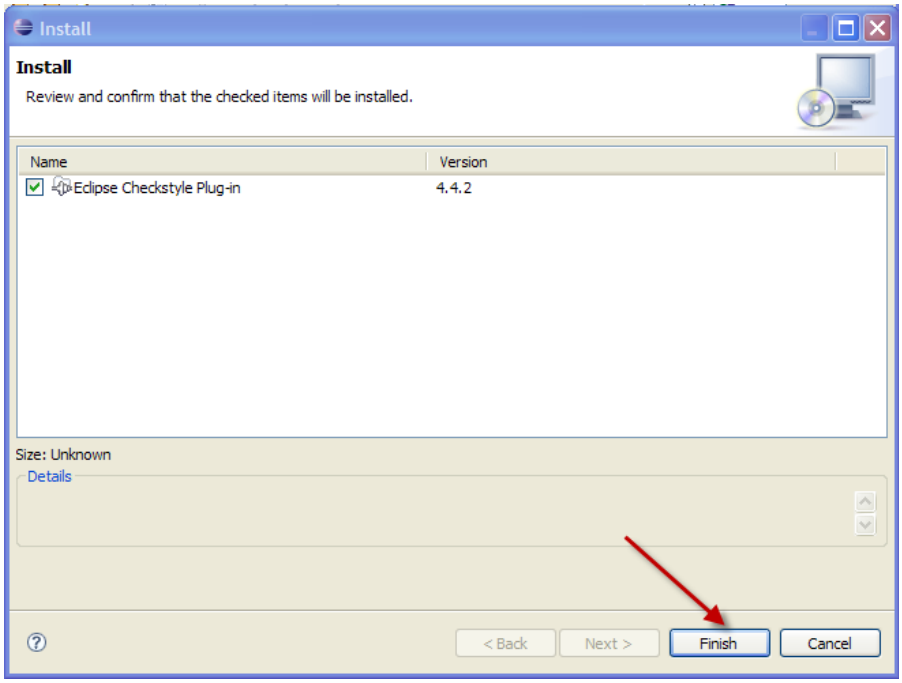
2. Ajouter un nouveau site d'installation avec l'URL : *http://eclipse-cs.sf.net/update/*



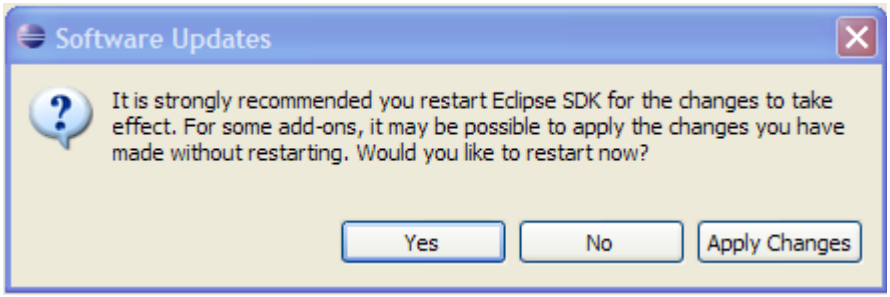
3. Choisir la version du plugin à installer (version 5-5). Installer.



4. Confirmer l'installation des plugins à installer



5. Relancer Eclipse



Avec Maven

```
<plugin>
	<groupId>org.apache.maven.plugins</groupId>
	<artifactId>maven-checkstyle-plugin</artifactId>
	<version>2.2</version>
	<configuration>
	<configLocation>config/sun_checks.xml</configLocation
	</configuration>
</plugin>
```

Configuration

Aller dans : Windows > Preferences > Checkstyle

Importer les règles de vérification en cliquant sur "New ...". Le fichier de "qualité" doit se trouver dans src/main/resources. Le fichier à importer est "checkstyle.xml"
Choisir ensuite "Set as default" pour utiliser les règles de vérification par défaut.

Utilisation

Par défaut, tous les projets sont automatiquement vérifiés. Vous pouvez désactiver checkstyle sur un projet en selectionnant le projet avec click droit > CheckStyle > Desactivate Checkstyle"

Sur un fichier du projet, vous pouvez faire : Click Droit > Checkstyle > Check code with checkstyle. Un rapport est généré avec toutes les violations de règles de vérification.

Avant chaque commit, vérifier chaque classe en attente de commit avec Checkstyle et résoudre chaque partie/portion de code qui est lève une violation.

Mise en application

Les différents modules exprimés dans le fichier de configuration XML de Checkstyle vont permettre de définir vérifications à effectuer sur le projet.

```
<module name="Checker">
   <module name="JavadocPackage"/>
   <module name="TreeWalker">
      <module name="AvoidStarImport"/>
      <module name="ConstantName"/>
      <module name="EmptyBlock"/>
   </module>
</module>
```

Le module “JavadocPackage” va aller vérifier que tous les packages ont une documentation liée.
Le module TreeWalker a lui même trois sous-module de vérification qui vont aller vérifier respectivement :

- La non-présence d’import de package finissant par “.*”
- Un nom de classe valide et invariant
- La non-présence de block “{ }” vide dans la classe

PMD

Principe

PMD va analyser le code afin de trouver des problèmes potentiels tel que :

- Bugs possibles,
- Code mort,
- Code non optimal,
- Expressions trop compliquées,
- Problèmes de sécurité,
- Problèmes de couplage entre objet/package

Procédure d'installation (plugin)

Avec Eclipse Installer

Similaire à celle de CheckStyle mise à part les points suivants :

2.Ajouter un nouveau site d'installation avec l'URL :
http://pmd.sourceforge.net/eclipse

Avec Maven

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-pmd-plugin</artifactId>
   <version>2.4<version>
   <configuration>
      <rulesets>
         <ruleset>/rulesets/basic.xml</ruleset>
         <ruleset>/rulesets/imports.xml</ruleset>
         <ruleset>/rulesets/unusedcode.xml</ruleset>
         <exclude name="UnusedPrivateField" />
         <ruleset>/rulesets/finalizers.xml</ruleset>
      </rulesets>
   </configuration>
</plugin>
```

La liste des "ruleset" de pmd se trouve à l'adresse suivante :
http://pmd.sourceforge.net/rules/index.html

Une fois la liste des ruleset définie, toutes les classes du projet seront parcourrue et un rapport en sera ressorti sous forme TEXT, XML ou bien HTML.

Configuration

Aller dans Windows > Preferences > PMD > Configuration des règles

Le fichier à importer est "pmd.xml"

Utilisation

Sur un fichier du projet, vous pouvez faire : Click Droit > PMD > Check code with PMD.

Un rapport est généré avec toutes les violations de règles de vérification.

Avant chaque commit, vérifier chaque classe en attente de commit avec PMD et résoudre chaque partie/portion de code qui est lève une violation.

Mise en application

Exemple du ruleset Design concernant la règle : AvoidReassigningParameters

Ce code lèvera un "Warning" sur la ruleset Design

```
public class Foo {
  private void foo(String bar) {
    bar = "something else";
    // Code
  }
}
```

Il est préférable de réassigner une nouvelle variable locale plutôt que d'utiliser celle passée en paramètre (même si l'exemple de cette règle est discutable dans certains cas)

```
public class Foo {
	private void foo(String bar) {
		String barLocale = "something
else";
		// Code
	}
}
```

Exemple du ruleset Clone concernant la règle : ProperCloneImplementation

Ce code lèvera un "Warning" sur la ruleset Clone

```
class Foo{
	public Object clone(){
		return new Foo(); // This is a bad use
	}
}
```

Il est préconisé que la classe implémente l'interface "Clonable" et que la méthode "parent" soit appellée au lieu de retourner une nouvelle instance de la classe Foo comme réalisé précédemment.

```
class Foo implements Clonable{
	public Object clone() throws CloneNotSupportedException{
		return (Foo) super.clone();
	}
}
```

Exemple du ruleset Unused Code concernant la règle : UnusedLocalVariable

```
public class Foo {
	public int doSomething() {
		int i = 5; // Unused
		return 2;
	}
}
```

```
public class Foo {
	public int doSomething() {
		int i = 5; // Used
		return i;
	}
}
```

Findbugs

Principe

Findbugs va trouver les bugs potentiels en analysant le bytecode Java. Pour cela il s'appuie sur une notion de 'bug patterns'.

Ces bugs sont classés en plusieurs catégories :

- Correctness

    - Regroupe les bugs généraux. Par exemple les boucles infinies, mauvaises utilisations de equals(), ...
- Bad practice
    - Regroupe les mauvaises pratiques. Par exemple les problèmes d'Exception, de ressources non fermées, mauvaises utilisations de comparaison de chaîne de caractères, ...
- Performance
    - Regroupe les problèmes de performance. Par exemple la création d'objets inutiles.
- Multithreaded correctness
    - Regroupe les problèmes liés au code multithread.
- Internationalization
    - Regroupe les problèmes liés à l'internationalisation d'une application.
- Malicious code vulnerability
    - Regroupe les problèmes de vulnérabilité. Par exemple du code qui pourrait être détourné de son utilisation, ...
- Security
    - Regroupe les problèmes de sécurité. Par exemple les problèmes liés au protocole http, les SQL injections, ...
- Dodgy
    - Regroupe le "smell code". Par exemple les comparaisons redondantes avec null, variables non utilisées, ...

Il existe un autre paquet de règles.
On peut le trouver au format jar sur http://fb-contrib.sourceforge.net/

Procédure d'installation (plugin)

Avec Eclipse Installer

Similaire à celle de CheckStyle mise à part les points suivants :

2.Ajouter un nouveau site d'installation avec URL :
http://findbugs.cs.umd.edu/eclipse/

Avec Maven

```
<plugin>
	<groupId>org.codehaus.mojo</groupId>
	<artifactId>findbugs-maven-plugin</artifactId>
	<version>1.0.0<version>
</plugin>
```

Des filtres peuvent être définis pour chaque Classe ou type de classe (suivant un pattern) avant de réaliser des vérifications distinctes.

Configuration

Aller dans Windows > Preferences > Java > Findbugs > Filter Files.
Le fichier à importer est “findbug.xml”

Utilisation

Sur un fichier du projet, vous pouvez faire : Click Droit > Findbugs > Findbugs.

Un rapport est généré avec toutes les violations de règles de vérification.

Avant chaque commit, vérifier chaque classe en attente de commit avec Findbugs et résoudre chaque partie/portion de code qui est lève une violation.

Mise en application

Trois exemples de filtres (par classe, par catégorie de règle de vérification et par Class Pattern)
La classe MyClass du package “com.foobar” sera analysée pour les règles de vérification “DE, UrF, SIC”

```
<Match>
	<Class name="com.foobar.MyClass"/ >
	<Bug code="DE,UrF,SIC" />
</Match>
```

Toutes les classes seront analysées pour les règles de vérification de la catégorie “PERFORMANCE”

```
<Match>
	<Bug category="PERFORMANCE" />
</Match>
```

Toutes les classes finissant par *Test seront analysées pour les règles de vérification excluant “IJU”

```
<Match>
	<!-- the Match filter is equivalent to a logical 'And' -->
	<Class name="~.*\.*Test" />


	<!-- test classes are suffixed by 'Test' -->
	<!-- 'IJU' is the code for bugs related to JUnit test code -->

	<Not>
		<Bug code="IJU" />
	</Not>
</Match>
```

Exemple complet

```
<FindBugsFilter>
	<Match>
		<Class name="com.foobar.ClassNotToBeAnalyzed" />
	</Match>

	<Match>
		<Class name="com.foobar.ClassWithSomeBugsMatched" />
		<Bug code="DE,UrF,SIC" />
	</Match>

	<!-- Match all XYZ violations. -->
	<Match>
		<Bug code="XYZ" />
	</Match>

	<!-- Match all doublecheck violations in these methods of
"AnotherClass". -->
	<Match>
```

```
            <Class name="com.foobar.AnotherClass" />
            <Or>
            <Method name="nonOverloadedMethod" />
            <Method name="frob" params="int,java.lang.String"
returns="void" />
            <Method name="blat" params="" returns="boolean" />
            </Or>
            <Bug code="DC" />
      </Match>

      <!-- A method with a dead local store false positive (medium priority).
-->
      <Match>
            <Class name="com.foobar.MyClass" />
            <Method name="someMethod" />
            <Bug pattern="DLS_DEAD_LOCAL_STORE" />
            <Priority value="2" />
      </Match>

      <!-- All bugs in test classes, except for JUnit-specific bugs -->
      <Match>
            <Class name="~.*\.*Test" />
            <Not>
                  <Bug code="IJU" />
            </Not>
      </Match>
</FindBugsFilter>
```

Historique des versions

VERSION	DATE	AUTEUR	DESCRIPTION	STATUT
1.0	11/01/2012	PERUCCA Jonathan		
0.1	10/01/2012	PERUCCA Jonathan		