amU Faculté des sciences Aix Marseille Université

Année 2025-26

Domaine Sciences et Technologies
Master informatique
Génie logiciel : TP 4
Codes UE : SINA09EL
Tests : analyse statique de code

Dans ce TP, nous allons étudier quelques outils pour analyser statiquement des programmes. Nous allons étudier le comportement de ces outils sur des programmes exemples pour évidemment y déceler des bugs ou des failles potentielles, mais également pour permettre de tester certaines limites ou restrictions de ces outils.

Le premier de ces outils sera l'outil *Checkstyle*, le second sera l'outil *SpotBugs*, le troisième sera l'outil *PMD* et le dernier l'outil *SonarLint*. Nous utiliserons également l'outil intégré à IntelliJ.

Le moteur de production *gradle* a été configuré pour utiliser les 4 outils décrits dans ce TP mais il est aussi possible de les utiliser en ligne de commande ou en tant que plugins dans IntelliJ.

1 Outil Checkstyle

Checkstyle est un outil d'analyse statique de code Java qui se concentre principalement sur le style de codage. Il permet de vérifier si le code respecte certaines conventions de style. Il est possible de définir ses propres règles de style via des fichiers *xml*.

Il y a plusieurs manières d'utiliser *Checkstyle* :

— En ligne de commande, en téléchargeant le jar exécutable depuis le dépôt officiel. On peut alors lancer l'analyse d'un fichier Java avec la commande suivante :

```
java -jar checkstyle-13.1.0-all.jar -c config/checkstyle/checkstyle.xml MonFichier.java
```

où `-c` permet de spécifier le fichier de configuration des règles à utiliser (ici les règles Google présentes dans le répertoire `TP4`).

— Via un plugin IntelliJ IDEA, disponible sur le marketplace des plugins IntelliJ (`settings > Plugins`) et d'installer le plugin `CheckStyle-IDEA`. On peut ouvrir la fenêtre *Checkstyle* via le menu `View>Tool Windows>Checkstyle` et lancer l'analyse en cliquant sur l'icône de lecture (play).

— via gradle en utilisant le plugin checkstyle déjà configuré avec les tâches gradle `checkstyleMain` (code principal) et `checkstyleTest` (code de tests). Cette analmyse utilise également les règles Google indiqué dans le fichier `config/checkstyle/checkstyle.xml` présent dans le répertoire `TP4`).

Question 1 : Lancez l'analyse de *Checkstyle* avec les règles *Google checks* sur le fichier `Complex.java` présents dans le répertoire `TP4/src/main/java` et étudiez les résultats produits. Quels sont les types de problèmes détectés par *Checkstyle* ?

Question 2 : Faites des corrections aux problèmes relevés par *Checkstyle* mais sans essayer d'obtenir zéro problèmes. Afin de garder un historique des modifications, créez une copie du fichier `Complex.java` dans un package `checkstyle` avant de les modifier. Quels sont les types de problèmes encore remontés par *Checkstyle* ?

2 L'analyseur SpotBugs et l'inspection IntelliJ

SpotBugs est un linter de programmes Java. Il s'agit d'un fork du projet *FindBugs*, qui n'est plus aujourd'hui maintenu. Ils sont l'un comme l'autre libres de droit.

SpotBugs comme de nombreux linters définit et utilise des motifs de bug et réalise donc une analyse approchée. Il est donc possible que certains bugs signalés n'en soit pas vraiment. Il s'agit de "faux positifs". Il est donc possible que, malgré une écriture soignée du code, certains bugs signalés ne puissent pas être éliminés.

SpotBugs classe les bugs trouvés selon plusieurs catégories, à savoir "Bad Practice", "Correctness", "Experimental", ...

> **Question 3 :** À quoi correspondent ces différentes classes ? Vous pouvez vous reporter au manuel pour répondre à cette question et avoir plus de détails sur les bugs pouvant être détectés.

2.1 Installation dans IntelliJ IDEA

On peut installer *SpotBugs* comme un plugin de l'IDE IntelliJ : Il suffit de choisir le menu et onglet **Preferences/Plugins**. Sur le **Marketplace**, rechercher *SpotBugs* et installez-le.

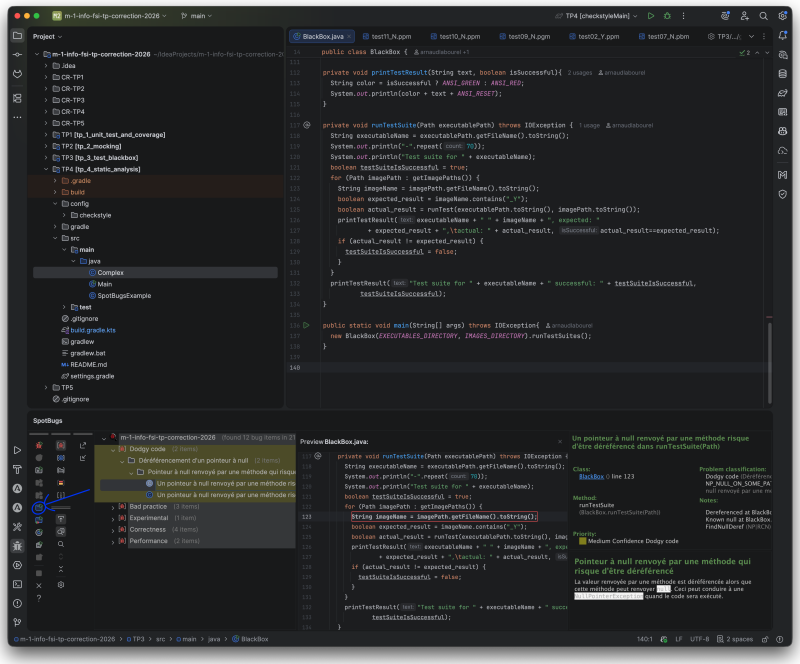
2.2 Utilisation de *SpotBugs* sur quelques exemples

Nous allons considérer quelques programmes Java et étudier les résultats de leur évaluation par *SpotBugs*.

Attention, *SpotBugs* travaille en réalité sur le *bytecode* Java, il ne fonctionne donc qu'une fois le code compilé. Il est nécessaire de bien recompiler après chaque modification dans le code.

2.3 Utilisation de *SpotBugs* dans IntelliJ IDEA

Ouvrez le fichier `Complex.java` dans IntelliJ IDEA. Ouvrir la fenêtre *SpotBugs* (`View>Tool Windows>SpotBugs`) et lancer l'analyse en cliquant sur l'icône de lecture (play).



Question 4 : Identifiez les bugs soulevés par analyse et expliquez leur cause.
Dans la méthode `infinite`, modifier l'appel récursif :

- en le gardant d'une condition `if (true)`,
- en le gardant d'une condition `if (! true)`,
- en le gardant d'une condition `if (a)` où `a` est précédemment déclaré comme étant vrai,
- en le gardant d'une condition `if (! a)` où `a` est précédemment déclaré comme étant vrai,
- en le remplaçant une boucle de type `while (true)`.
- en le remplaçant une boucle de type `while (b)` où `b` est précédemment déclaré comme étant vrai.

Que constatez-vous dans chacun des cas ?

Question 5 : Proposez des corrections aux bugs restants permettant de passer l'analyse sans encombre. Afin de garder un historique des modifications, créez une copie du fichier `Complex.java` dans un package `spotbugs` avant de le modifier.

2.4 Utilisation de l'analyseur intégré à IntelliJ IDEA

IntelliJ possède son propre outil d'analyse configurable sous `Preferences>Editor>Inspections`. Ses résultats se matérialisent la plupart du temps comme des avertissements (*warnings*). On peut y accéder soit via une icône en haut à droite de la fenêtre d'édition, sous via le menu `View>Tools Window>Problems`.

Question 5 : Étudier les résultats de l'inspection et tout particulièrement, les deux derniers messages et tenter de les corriger.

Question 6 : En comparant les alertes levées par *SpotBugs* et celles levées par `Inspections` et en considérant des artéfacts tels que le graphe de flot de contrôle, tentez de discuter la manière de fonctionner de ces deux outils.

2.5 Utilisation de *SpotBugs* sur des exemples plus complexes

Importer dans IntelliJ le fichier `SpotBugsExample.java` et lancer l'analyse.

Question 7 : Identifiez les bugs soulevés par analyse *SpotBugs* et expliquez leur cause.

Question 8 : Comparez ces bugs avec ceux également signalés par l'inspection IntelliJ.

Question 9 : Proposer des corrections aux bugs signalés par *SpotBugs* permettant de passer son analyse sans encombre. Placer une copie du fichier `SpotBugsExample.java` dans un package `spotbugs` avant de le modifier.

Importer dans IntelliJ le fichier `Main.java` et lancer l'analyse.

Question 10 : En se focalisant sur les bugs de la classe "Correctness", inspecter les bugs relevés par *SpotBugs* et identifiez la cause. Existe-t-il ailleurs dans le code un bug de type similaire non signalé par *SpotBugs* ?

Inspectez maintenant la version décompilée issue du *bytecode* de ce programme (obtenu en cliquant sur `Main` dans l'onglet `Project->out->production`. Selon vous, qu'est-ce qui justifie que le premier bug a été identifié et non le second ?

Question 11 : Comparez ces bugs avec ceux également signalés par l'inspection IntelliJ. Appliquez les simplifications proposées par l'inspection Java. Sont-elles correctes ?

L'outil *SpotBugs* identifie un grand nombre de bugs.

Question 12 : En vous aidant de la documentation, écrire un petit programme qui lève les bugs suivants :

- **(NP_EQUALS_SHOULD_HANDLE_NULL_ARGUMENT)**,
- **(NP_UNWRITTEN_FIELD)**,
- **(UWF_NULL_FIELD)**,
- **(RANGE_ARRAY_INDEX)**,
- **(RANGE_ARRAY_OFFSET)**.

Question 13 : En vous inspirant des questions précédentes, testez les limites de la détection en proposant si cela est possible des situations où la cause du bug est présente, mais l'outil ne permet pas de le détecter. Créez une ou plusieurs classes Java levant ces bugs dans le package `spotbugs`.

3 L'analyseur *PMD*

L'analyseur PMD permet notamment l'analyse de code Java. Il s'agit d'un programme "stand-alone" mais il existe un plugin IntelliJ qui permet de faire appel à ce programme qui produira alors ses rapports au sein de cet IDE. Contrairement à *SpotBugs* , *PMD* travaille sur le source Java du programme et est un outil fortement configurable puisque les règles servant à l'analyse sont définies extérieurement au logiciel dans un fichier xml. Cependant, nous n'entrerons pas plus avant dans ces fichiers et la possibilité de définir ses propres règles.

3.1 Installation

3.1.1 Installation locale de PMD

La méthode d'installation dépend du système d'exploitation.

3.1.1.1 Linux/MacOS

Le petit script suivant permet de faire une installation locale du logiciel :

```
cd $HOME
wget https://github.com/pmd/pmd/releases/download/pmd_releases%2F7.21.0/pmd-dist-7.21.0-bin.zip
unzip pmd-dist-7.21.0-bin.zip
alias pmd="$HOME/pmd-bin-7.21.0/bin/pmd"
```

3.1.1.2 Windows

Il est possible d'installer *PMD* sous Windows en suivant les instructions suivantes :

1. Télécharger pmd-dist-7.21.0-bin.zip
2. Extraire l'archive zip, par exemple dans `C:\pmd-bin-7.21.0`
3. Ajouter le dossier `C:\pmd-bin-7.21.0\bin` au `PATH`, soit :

— Définitivement : Via le dialogue Propriétés système > Variables d'environnement > Ajouter à la variable `PATH`
— Temporairement, en ligne de commande : `SET PATH=C:\pmd-bin-7.21.0\bin;%PATH%`

3.1.2 Utilisation

Ensuite, il est possible d'utiliser *PMD* en ligne de commande. Par exemple, pour analyser le contenu du répertoire `src/java/main` avec les règles de base de *PMD* (option `-R rulesets/java/quickstart.xml`) au format texte (option `-f text`). L'utilisation de *PMD* peut se faire de manière différente selon le système d'exploitation utilisé :

3.1.2.1 Linux/MacOS La ligne de commande est la suivante :

```
$ pmd check -d src/java/main -R rulesets/java/quickstart.xml -f text
```

3.1.2.2 Windows La ligne de commande est la suivante :

```
pmd.bat check -d c:\src -R rulesets/java/quickstart.xml -f text
```

3.1.3 Options de PMD

L'option `-d` permet de spécifier le répertoire à analyser, l'option `-R` permet de spécifier le fichier de règles à utiliser et l'option `-f` permet de spécifier le format du rapport produit. Plusieurs formats sont possibles, notamment `text`, `xml`, `html`, ...

Les fichiers de règles sont disponibles au lien suivant : docs.pmd-code.org/pmd-doc-7.21.0/pmd_rules_java.html

3.2 Utilisation de PMD dans IntelliJ IDEA

Il suffit d'installer le plugin `PMDPlugin` sous IntelliJ IDEA.

On peut lancer l'analyse PMD via le menu : `Tools>Run PMD>Pre defined>All`.

La fenêtre PMD s'ouvre et affiche les résultats de l'analyse. On peut aussi ouvrir la fenêtre via `View>Tool Windows>PMD`.

3.3 Utilisation de PMD sur des exemples

Question 14 : Reprenez les 3 fichiers Java de l'exercice précédent à la fois dans leur version originale et dans la version corrigée par vos soins et soumettez-les à *PMD*. Tentez ensuite d'améliorer les codes en essayant de minimiser les alertes PMD. Là aussi, créez une copie des fichiers originaux dans un package `pmd` avant de les modifier.

Question 15 : Quel est le résultat de l'analyse *PMD* sur les programmes que vous aurez proposés en réponse à l'exercice ?

4 L'analyseur SonarLint

SonarLint est un élément de la suite SonarQube, outil de mesure de qualité du code. Vous l'utiliserez directement via le moteur de production gradle avec les tâches `sonarlintMain` et `sonarlintTest`.

Question 16 : Reprenez les 3 fichiers Java des exercices précédents à la fois dans leur version originale et dans la version corrigée par vos soins et soumettez-les à *SonarLint*. Tentez ensuite d'améliorer les codes en essayant de minimiser les alertes *SonarLint* et en suivant ses recommandations. Là aussi, créez une copie des fichiers originaux dans un package `sonarlint` avant de les modifier.