Sonar fixer

Clement Maxime - Piorun Jordan

November 15, 2015

1 Introduction

Pour de nombreux projets, nous avons recours a Sonar, un logiciel permettant de mesurer la qualité de notre code en continu. Sonar classe les améliorations a apporter en différentes catégories, des erreurs critiques aux simples informations.

Pour notre projet, nous somme parti d'un simple constat : les erreurs sonar sont souvent les mêmes, et un nombre considérable d'entre elles pourrait être corrigées de façon automatique. Cela passe par deux étapes : premièrement il faut repérer le morceau de code qui est à l'origine de l'erreur, puis il faut ensuite le remplacer avec du code équivalent.

Pour ces deux besoins, nous avons choisis d'utiliser Spoon, une librairie développée à l'Inria permettant d'analyser et de transformer du code Java.

2 Analyse du code

2.1 La méthodologie

comment on a choisi nos regles -¿ choix en fonction des projets

Avant même de s'intéresser à la partie technique de ce projet, nous avons du établir la liste des erreurs que nous voulions corriger.

Sonar classe les erreurs en 5 catégories : bloquantes, critiques, majeures, mineures et informatives. Parmi celles ci, nous avons jugé que les plus pertinentes, mais aussi les plus urgentes à corriger sont les erreurs majeures.

En effet, les deux catégories au dessus gênent souvent le bon fonctionnement du programme et sont donc en toute logique détectée au préalable par le développeur. Dans le cas où le programme fonctionnerait malgré ces erreurs, elles sont souvent très peu nombreuses, et donc rapide à corriger.

Les niveaux de sévérité moins importants quant à eux relèvent plus de conventions de codages ou autres problèmes pouvant gêner la compréhension du code. Nous gardons à l'idée que comprendre un code facilement est essentiel pour le maintenir, mais nous préférons mettre la priorité sur la catégorie restante : les erreurs majeures.

En ce basant sur les tests unitaires, nos transformations ne devront pas faire régresser le code sinon la transformations ne seras pas considéré comme acceptable.

2.2 Les règles définies

Afin de choisir les différentes règles que nous corrigerons de manière automatique, nous avons pensé directement à l'évaluation. Nous avons fait analyser des gros projet open-source tel que Apache Commons-lang[1] et Apache Commons-math[2], et nous avons choisi notre ensemble d'erreur parmi celle relevé par Sonar.

Ces alertes Sonar proviennent de mauvaises pratiques de codage, qui ne gênent pas le bon déroulement du code, mais qui pour certaines d'entre elles, pourraient entrainer des erreurs lors d'ajout de code. Après cette première réduction du scope initial, nous avons fais le tri entre les alertes que nous pouvons corriger de façon automatique, et celles qui dépendent du contexte. Suite a cette seconde analyse, nous avons dressé le tableau en annexe 1 qui reprend les différentes alertes ainsi que la solution apportée.

2.3 Correction d'erreur

Afin de détecter les erreurs que nous voulons corriger, nous nous sommes appuyé sur la librairie Spoon, qui nous apporte une solution pour analyser du code Java. Pour chaque alerte, nous avons donc un une classe Processor associée dans laquelle nous décrivons le pattern à rechercher. Lorsque cette fonction trouve du code qui match, nous pouvons commencer sa transformation.

3 Évaluation

3.1 Les test unitaires

couverture de test et pourcentage de mutation (partie courte)

3.2 Projets réels

Dans le but de valider notre approche, nous avons décider de prendre des gros projets. Ces projets sont les suivants : Apache Commons-lang, Apache Commons-math. L'annexe 2 montre les résultats obtenus suite à nos transformation.

Les projets utilisés ne sont pas simplement les clones des répertoires GitHub respectifs. Pour chacun des projets nous avons choisi comme point de départ le résultat renvoyer par Spoon sans appliquer une seule transformation de notre part. En effet, le comportement de Spoon est de créer l'AST[3] represensant le code source, d'appliquer les transformations spécifié par le développeur, et pour finir de convertir l'AST en code source.

[1] `https://github.com/apache/commons-lang`
[2] `https://github.com/apache/commons-math`
[3] Abstract Synthax Tree

Durant ce processus Spoon est susceptible de rajouter ou de perdre de l'information (ex : les parenthèses), c'est pourquoi nous avons choisi ce résultat comme point de départ.

4 Difficultés

Comme l'indique l'annexe 2, certaine de nos transformations ont provoqué une régression. Ces régressions peuvent s'explique de différentes manières.

4.1 Spoon

Pour Apache Commons-math, Spoon produit des NullPointer[4] lors de l'utilisation de Filter[5] pour une raison encore inconnu.

4.2 Comportement rechercher

Il est arrivait que le comportement des tests aillent à l'encontre des règles de Sonar. Par exemple, pour Apache Commons-lang, nous avons voulu supprimer les attributs inutilisés d'une classe. Cependant 5 tests concernant la réflectivité consistaient à accéder à ces champs de manières dynamiques. Il nous est donc impossible de prévoir ce comportement.

4.3 Passage par copie

Le but d'une des transformations est de réduire le nombre de ligne entre deux cases afin que ce nombre n'excède pas 5. Notre approche à était de créer des sous-méthodes regroupant le contenu du case afin de garder le comportement initial du programme. Pour que le code source reste cohérent et compilable, nous avons du récupérer tous le contexte avant l'exécution du case, et le passer en paramètre a notre sous-méthodes. Cependant, même une fois cette étape réaliser, nous avons pu observer de forte régression du code source. Cela s'explique à cause d'une des spécificité de Java, le passage de copie[6].

En effet, lors d'un appel de méthode, Java crée une copie de l'objet afin de le passer en paramètre. Si à l'intérieur de la méthode, nous modifions la valeur d'un paramètre (=), alors il n'y a pas d'effet de bord dans le case initial et le comportement initial n'est pas respecter. Cependant, si l'on modifie un attribut de l'objet passé en paramètre alors l'effet de bord est possible. La solution à donc était la suivante :

1. Parmis les variables du contexte, déterminer celles qui seront modifier dans la méthode.
2. Pour chacune d'entre elles,
   (a) Créer un objet qui les encapsuleras (attribut).
   (b) Remplacer le type par le nouveau.
   (c) Remplacer tout les accès par l'appel du Getter
   (d) Remplacer toute les affections par l'appel du Setter

Cependant, sûrement suite à un problème d'implémentation, cette solution produit de la régression lorsque des cases sont imbriqué dans l'AST (pas forcement directement).

[4] `http://docs.oracle.com/javase/8/docs/api/java/lang/NullPointerException.html`
[5] `http://spoon.gforge.inria.fr/filter.html`
[6] `https://docs.oracle.com/javase/tutorial/java/javaOO/arguments.html`

A Annexe 1 - Tableaux des correctifs

Noncompliant	Compliant

`class Bar {`
`  public boolean doSomething(){...}`
`}`
`class Foo extends Bar {`
`  public boolean doSomething(){...}`
`}`
	`class Bar {`
	`  public boolean doSomething(){...}`
	`}`
	`class Foo extends Bar {`
	`  @Override`
	`  public boolean doSomething(){...}`
	`}`

`switch (myVariable) {`
`  case 0: // 6 lines till next case`
`  methodCall1("");`
`  methodCall2("");`
`  methodCall3("");`
`  methodCall4("");`
`  break;`
`  case 1:`
`  ...`
`}`
	`switch (myVariable) {`
	`  case 0:`
	`  doSomething()`
	`  break;`
	`  case 1:`
	`  ...`
	`}`
	`...`
	`private void doSomething(){`
	`  methodCall1("");`
	`  methodCall2("");`
	`  methodCall3("");`
	`  methodCall4("");`
	`}`

`public class MyClass {`
`  private int foo = 42;`
 	
`  public int compute(int a) {`
`  return a * 42;`
`  }`
`}`
	`public class MyClass {`
	`  public int compute(int a) {`
	`  return a * 42;`
	`  }`
	`}`

`if ( "foo".equals(bar) ){`
`  ...`
`}`
	`if ( bar.equals("foo") ){`
	`  ...`
	`}`

B Annexe 2 - Évaluation

	Alerte ID	Alertes	Alertes Fixés	% Alertes Fixés	Test degradé
Commons-lang	ID_override	2	2	100%	0
	ID_switch	21	21	100%	3
	ID_private_field	4	4	100%	5
	ID_equals	17	17	100%	0
Commons-math	ID_override	9	9	100%	0
	ID_switch	61	N/A	N/A	N/A
	ID_equals	9	9	100%	0