Sonar fixer

Clement Maxime - Piorun Jordan

November 16, 2015

Contents

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 sommes parti d'un simple constat : les erreurs sonar sont souvent les mêmes, et un nombre considérable d'entres elles pourraient 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

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

Sonar classe ces erreurs en 5 catégories : bloquantes, critiques, majeures, mineures et informatives. Parmi celles ci, notre attention s'est tout particulièrement portée sur les erreurs majeures. En effet, elles semblent être les plus rependues dans les projets, et peuvent pour certaines d'entre elles rendre un projet difficilement maintenable.

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 à cette première approche du problème nous nous sommes rapidement heurté à un problème : comment tester nos résultats ? L'idée initiale était de sélectionner des projets open source, que nous récupérons sur GitHub, afin que sonar puisse effectuer un premier diagnostic. Notre programme s'occupe ensuite de corriger les erreurs que nous avons ciblé, et un deuxième diagnostic permet de confirmer que le nombre d'alerte a effectivement baissé.

Cependant, il est difficile de trouver des projets qui contiennent des erreurs bien spécifiques, d'autant plus que certaines d'entre elles relèvent de pratiques si mauvaises qu'il est très peu probable d'en croiser. Nous avons donc changé notre raisonnement en choisissant d'abord des projets que nous jugeons fiable, avec une communauté importante et une taille conséquente. Nous l'avons donné à analyser auprès de sonar, et nous avons choisis des alertes pertinentes parmi celles relevées. Au terme de cette analyse, nous avons dressé le tableau en "Annexe 1 - Tableaux des correctifs" qui reprend les différentes alertes ainsi que la solution apportée.

Cette seconde approche, basée sur les projets Apache Commons-Lang[1] et Apache Commons-math[2], nous a permis de dresser une nouvelle liste d'erreurs dont nous voulons automatiser la réparation.

2.2 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 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 Projets réels

Comme décrit précedemment, nous avons choisis de valider nos résultats sur des projets de taille imposantes. Ces derniers ne sont pas simplement les clones des répertoires GitHub respectifs. Pour chacun des projets nous avons choisi comme point de départ le résultat renvoyé par Spoon sans appliquer une seule transformation de notre part. En effet, le comportement de Spoon est de créer l'AST[3] représentant le code source, d'appliquer les transformations spécifiées par le développeur, et pour finir de convertir l'AST en code source. Durant ce processus Spoon est susceptible de rajouter ou de perdre de l'information (ex : les parenthèses), c'est pourquoi nous avons choisis ce résultat comme point de départ.

4 Difficultés

Comme l'indique "Annexe 2 - Évaluation", certaines de nos transformations ont provoqué une régression du code. Ces régressions peuvent s'expliquer de différentes manières.

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

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 Cas particuliers

Il est arrivé que les comportements des tests aillent à l'encontre des règles de Sonar. Par exemple, pour Apache Commons-lang, nous avons voulu supprimer les attributs privés 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 lignes entre deux cases afin que ce nombre n'excède pas 5. Notre approche à été 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-méthode. Cependant, même une fois cette étape réalisée, 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 voulu n'est pas respecté. Cependant, si l'on modifie un attribut de l'objet passé en paramètre, alors l'effet de bord est possible. La solution à donc été la suivante :

1. Parmi 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 tous 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 imbriqués dans l'AST (pas forcement directement).

5 Perspective

Afin de palier au problème important que constitue la régression de code, il serait intéressant d'envisager un outil graphique qui viendrait se positionner entre Sonar et le développeur. Ce dernier pourrait récupérer les rapports Sonar et ainsi ce passer de détection. Il serait également intéressant de pouvoir laisser le choix au développeur d'appliquer ou non une transformation, car ce dernier est sensé être au courant du comportement nominal du programme.

6 Conclusion

Notre but était de prouver qu'il est possible d'automatiser la résolution de certaines erreurs levées par sonar. Dans notre tâche, nous avons observé différentes limites telles que les règles qui dépendent d'un contexte particulier, ou les comportement qui vont volontairement à l'encontre des règles établies par Sonar.

Nous sommes convaincu qu'un bon nombre d'alertes pourraient encore se corriger à l'aide de notre programme en écrivant de nouveaux processeurs.

[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`

7 References

1. Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, Lionel Seinturier. “Spoon: A Library for Implementing Analyses and Transformations of Java Source Code”. http://spoon.gforge.inria.fr/

2. Lien github du projet : `https://github.com/maxcleme/OPL-Rendu1`

A Annexe 1 - Tableaux des correctifs

Noncompliant	Compliant

squid:S1161 - @Override annotation should be used on any method overriding
 	
`class Bar {`
`  public boolean doSomething(){...}`
`}`
`class Foo extends Bar {`
`  public boolean doSomething(){...}`
`}`
	squid:S1161 - @Override annotation should be used on any method overriding
	
	`class Bar {`
	`  public boolean doSomething(){...}`
	`}`
	`class Foo extends Bar {`
	`  @Override`
	`  public boolean doSomething(){...}`
	`}`

squid:S1151 - switch case clauses should not have too many lines
 	
`switch (myVariable) {`
`  case 0: // 6 lines till next case`
`  methodCall1("");`
`  methodCall2("");`
`  methodCall3("");`
`  methodCall4("");`
`  break;`
`  case 1:`
`  ...`
`}`
	squid:S1151 - switch case clauses should not have too many lines
	
	`switch (myVariable) {`
	`  case 0:`
	`  doSomething()`
	`  break;`
	`  case 1:`
	`  ...`
	`}`
	`...`
	`private void doSomething(){`
	`  methodCall1("");`
	`  methodCall2("");`
	`  methodCall3("");`
	`  methodCall4("");`
	`}`

squid:S1068 - Unused private fields should be removed
 	
`public class MyClass {`
`  private int foo = 42;`
 	
`  public int compute(int a) {`
`  return a * 42;`
`  }`
`}`
	squid:S1068 - Unused private fields should be removed
	
	`public class MyClass {`
	`  public int compute(int a) {`
	`  return a * 42;`
	`  }`
	`}`

squid:S1132 - Strings literals should be placed on the left side when checking for equality
 	
`if ("foo".equals(bar)){`
`  ...`
`}`
	squid:S1132 - Strings literals should be placed on the left side when checking for equality
	
	`if (bar.equals("foo")){`
	`  ...`
	`}`

B Annexe 2 - Évaluation

	Squid	Alertes	Alertes Fixés	% Alertes Fixés	Test degradé
Commons-lang	squid:S1161	2	2	100%	0
	squid:S1151	21	21	100%	3
	squid:S1068	4	4	100%	5
	squid:S1132	17	17	100%	0
Commons-math	squid:S1161	9	9	100%	0
	squid:S1151	61	N/A	N/A	N/A
	squid:S1132	9	9	100%	0