Université Lille 1

Rapport OPL

Sujet Libre

Outil de détection et d'éducation pour la défense contre les AntiPattern

Auteur :
Hedi Mokhtar

Responsable :
Martin Monperrus

31 mars 2017

Table des matières

Introduction

Le monde du logiciel peut être vu comme un monde artistique, les développeurs les plus expérimentés seront capable de distinguer du beau code de mauvais code, ou bien de voir des architectures agréables et rapides à arpenter plutôt que d'autres incompréhensibles. Le fait de prendre conscience de cette différence implique que les développeurs initiés aient réussi à intégrer un schéma d'analyse de code instinctif suite à un enseignement de cet art, ou bien suite à une succession de mauvaises expériences de développement qui les auront guidé vers une évolution.

Le sujet de code plus propre est énormément abordé et exploité, que ce soit dans des ouvrages qui nous mettent en garde contre les bad smells et leurs influences sur la dette technique [1], où nous montrent ce que sont les design pattern [2]. Il y a aussi tout un tas de logiciel découlant de ces savoirs déjà capable de corriger nos bad smells et générer de belles structures automatiquement. Le problème qui se pose ici est que la correction automatique a peu de chance d'aider une personne à se rendre compte de ses propres erreurs. Il y a t'il un moyen pour qu'un logiciel puisse faire plonger dans une réflexion son utilisateur, afin d'améliorer sa qualité de code à long terme en pointant du doigt ces AntiPattern [3] ?

Pour réaliser cela nous allons avoir besoin de deux parties distinctes : Tout d'abord des idées d'AntiPattern à détecter pour repérer l'erreur de l'utilisateur ainsi qu'une manière d'apprendre à l'utilisateur comment corriger ce défaut. Ce que nous souhaitons transmettre avec ce rapport est qu'il existe un moyen de trouver les erreurs de conception de logiciel au sein d'un programme et de former la personne responsable de ces erreurs à long terme.

Pour atteindre cet objectif, nous avons réfléchi à plusieurs algorithmes divers en recherchant, en innovant et en utilisant nos connaissances sur les Anti Pattern et Design Pattern pour les traquer et donner des instructions pour les corriger.

1. Expression apparue et souvent expliquée dans Clean Code de Robert Cecil Martin
2. Par exemple Head First Design Patterns d'Elisabeth Freeman
3. Contraire d'un design pattern, c'est à dire une structure qui nuit à la qualité de code et favorise la dette technique

Chapitre 1

Delta avec le rendu précédent

Ce rapport commencera par lister les points qui ont été changé pour ce dernier rendu. Ces points sont liés directement aux remarques qui ont pu être faites sur ce projet.

1.1 Re-factorisation

Le projet à été retravaillé pour une meilleur qualité de code. Ceci a été fait par le remaniement le modèle et les structures de données pour pouvoir être réutilisables. La duplication de code qui était du à un projet fait dans la hâte à été réduite.

1.2 Test

L'absence de test dans le rendu précédent rendait certain Handler incohérent. Des tests ont été implémenté pour chaque Handler. Pour tester des fonctions invoquant Spoon il aura fallu créer un projet de test ou chaque Handler à sa propre classe crée pour être évaluer et voir si un Handler attrape bien le cas de situation qu'il devrait.

1.3 Documentation et implémentation

Suite au travail de test les handler ont été modifié en conséquence et sont désormais opérationnel. De ce fait les travaux ont pu être adapté à un projet plus gros que ceux du rendu précédent afin d'être évalué il s'agit ici des sources d'Apache.

Chapitre 2

Travail technique

2.1 But

2.1.1 Définition

Définissons tout d'abord la notion clé de ce rapport, les AntiPattern sont des erreurs courantes de conception des logiciels. Ce nom vient du fait que ces erreurs sont apparues dès les phases de conception du logiciel, notamment par l'absence ou la mauvaise utilisation des design pattern. L'important à retenir est qu'ils engendrent tous une dette technique et sont synonyme de casse tête pour les développeurs qui ont ensuite à relire le code par dessus.

2.1.2 objectifs

Le premier objectif de notre travail est d'expérimenter diverses manière de repérer un type d'AntiPattern, des descriptions d'AntiPattern se feront un peu plus loin dans ce rapport. Pour faire cela nous avons dû expérimenter plusieurs algorithmes qui se basent sur un contexte différent car les AntiPattern ont peu de chose en commun pour la plupart. Ensuite si un AntiPattern à été trouvé on enchaîne avec une aide à la correction de celui ci.

2.2 Technologies

2.2.1 Langage

Le langage utilisé pour l'implémentation des algorithmes est le Java parce que c'est le langage objet le plus utilisé et que nos connaissances de design pattern et

d'AntiPattern sont vouées à celui ci.

Pour l'aide à la correction de l'erreur de conception de l'utilisateur, une interface Web est plus sympathique et pour une première version une simple page HTML expliquant l'erreur sera mise en place.

2.2.2 Outil

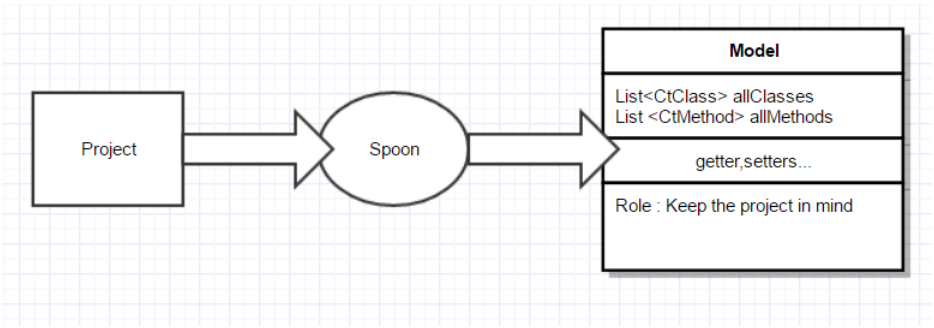
Afin de parcourir le projet de notre utilisateur pour y trouver les esquisses d'AntiPattern nous allons utiliser Spoon qui permettra de récupérer ce que l'on veut à la granularité que l'on souhaite.

2.3 Architecture

L'architecture de notre projet repose sur trois composants.

2.3.1 Le modèle

Le modèle s'occupe de récupérer toute les données du projet. Lorsque l'on analyse le projet on récupère toute les données qui nous semble intéressante une fois seulement. Le fait de le faire qu'une seule fois et de tout mettre dans notre propre structure de donnée permet de ne plus utiliser Spoon pour parcourir tout le projet à nouveau car il ne restera plus qu'a lancer nos Handlers d'AntiPattern sur notre modèle. . Les données utiles sont par exemples les suivantes : Toute les classes du projet, toute les méthodes du projets, toutes les variables et les déclarations de nouveaux objets, les versions des dépendances...

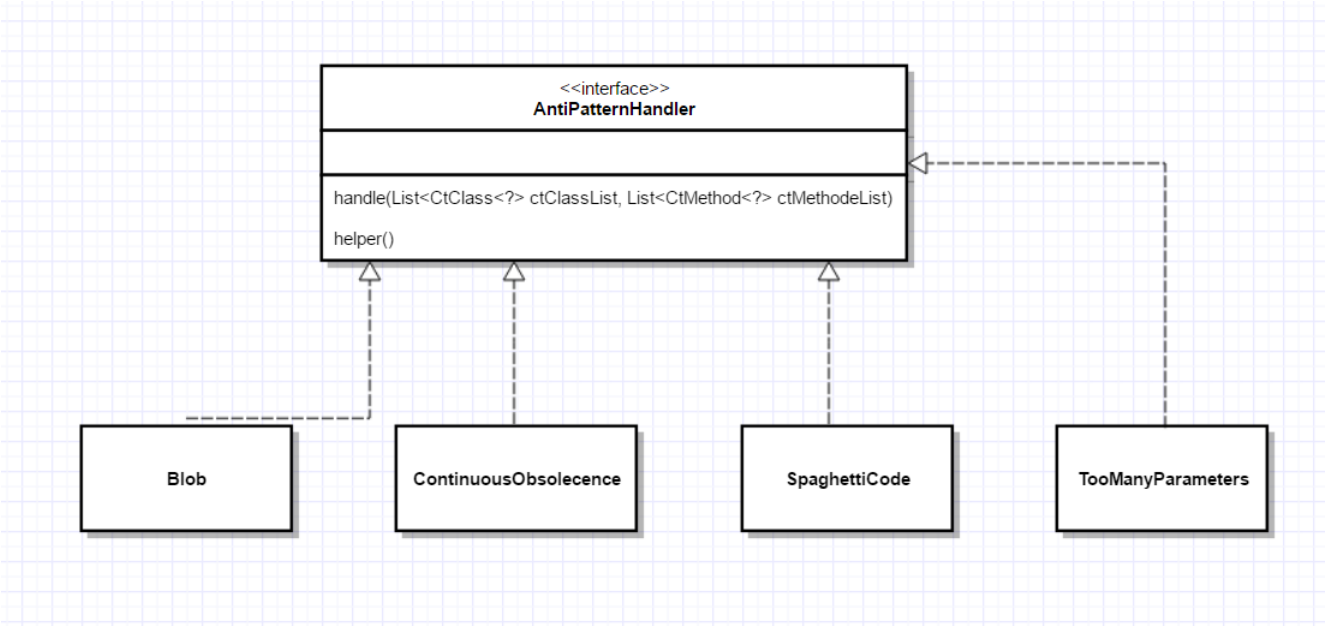


2.3.2 Les handler

Les handlers sont les algorithmes qui vont prendre en charge un cas d'AntiPattern en particulier, il y a beaucoup d'AntiPattern existants. L'algorithme d'un handler

lui est unique étant donné que les anti-pattern ne se ressemblent pas en général. La seule action commune à tout les Handlers est que lorsque ça fonction de détection d'AntiPattern il active un helper sur l'élément trouvé et le signal à l'utilisateur.

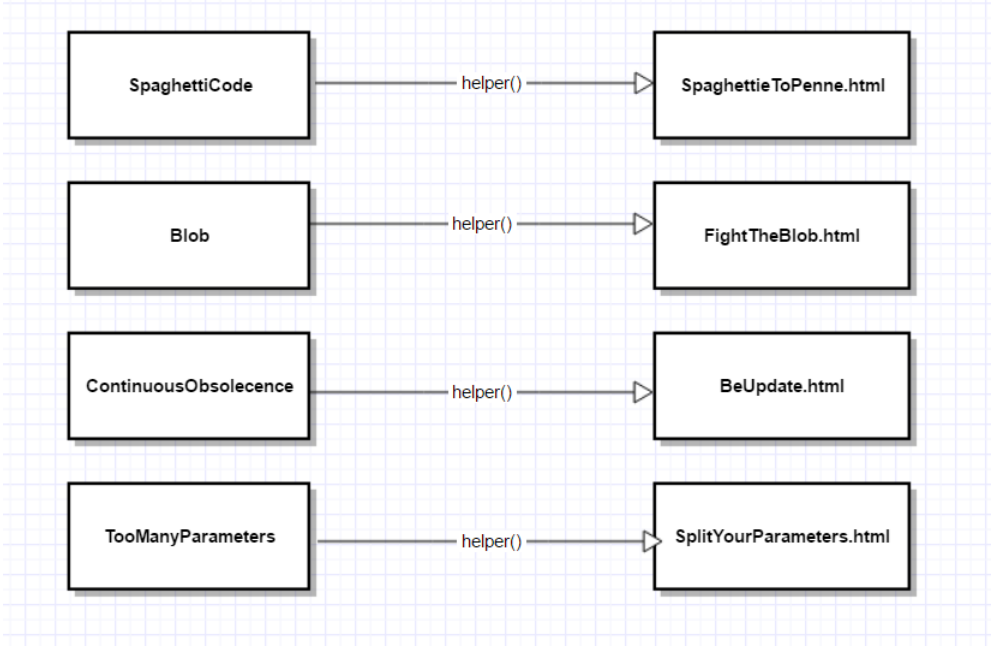
Voici un UML avec de la structure avec quelques AntiPattern



2.3.3 Les Helpers

Les Helpers représentent l'interface que les utilisateurs verront suite à la déclaration de la présence d'un AntiPattern, pour une première version une page html simple suffit pour expliquer le problème et comment le résoudre.

Voici une illustration abstraite de comment les Helpers sont appelés.



2.4 Algorithmes des AntiPattern

Pour trouver l'algorithme d'un AntiPattern c'est comme chercher à résoudre une énigme, pour certains c'est simple, pour d'autres moins et pour quelques uns on se dit même que cela est impossible, nous verrons ici comment raisonner pour trouver la présence d'AntiPattern dans un projet. Chaque algorithme utilise le modèle de donnée décrit dans la partie précédente.

2.4.1 Algorithmes simples

Les algorithmes présentés dans cette partie sont simples car de notre point de vue humain il est déjà facile de voir instinctivement la présence de l'AntiPattern, cela se rapproche du code smell où nous sommes capable de sentir une odeur venant du code.

1. Trop d'arguments

   Pour s'échauffer, dans cet algorithme, on regarde si le nombre de paramètres que requiert une fonction est raisonnable. La seule ambiguïté ici réside à déterminer le nombre maximum de paramètres avant que ce ne soit trop. La conséquence de cet AntiPattern est une difficulté de lecture et de test de la méthode par exemple.

Algorithm 1 TooManyParameters algorithm

```
1: procedure TooManyAlgorithme.handle(List(Methods)
2:     Pour toute les méthodes :
3:     if ParametersMethodNumber > MaxParametersAllowed then
         return this.Method.helper()
```

2. Paramètre Booléenne

   Un deuxième exemple simple à voir est la présence d'un paramètre de type Booléen qui dénonce le fait qu'une méthode fait deux choses à la fois.

2.4.2 Algorithme non triviaux à l'oeil nu

La différence majeur entre un AntiPattern et un Code Smell est que celui ci est difficile à voir, voici quelques exemples.

1. Code Spaghetti

   Le code Spaghetti représente du code qui nous force a lire tout le programme pour en comprendre le sens. Par exemple une méthode va en appeler une autre

à l'opposé du programme, qui en appel encore une autre et ainsi de suite. ici aussi la difficulté est de savoir au bout de combien d'appel de méthode cela devient du spaghetti.

Algorithm 2 SpaghettiCode algorithm

```
1: procedure SpaghettiCode.handle(List(Methods)
2:     Pour toute les méthodes :
3:     if la méthode appel une autre méthode dans une autre classe then
       return Recherche du même processus dans la nouvelle méthode en decrémentant en fonction de notre tolérance.
       Une fois cette méthode faites il est alors aussi intéréssant et possible d'obtenir l'appel d'enchaînement de méthode le plus long au sein du projet
```

2. Blob

   Le Blob est une classe plus imposante que les autres qui a toute les responsabilités, on ne se rend pas forcement compte de cette classe car si on la construit elle nous semble simple mais plus on lui donne de fonctionnalités, plus elle est susceptible d'en avoir de nouvelle et les autres classes à coté du Blob ne sont là que pour structurer des données et sont negligeables.

   L'algorithme d'un Blob est compliqué à définir, ici il a été décidé que si la taille d'une classe et nettement plus important que celle des autres alors c'est un Blob.

3. Obsolescence continue

   Les programmes qui ont du vécu ont souvent des dépendances ou une version de langage qui est obsolète, limités dans leurs vitesses et dans les habitudes de développement des programmeurs. Si on arrive à voir que la version du langage utilisé est très ancien, par exemple pour Java on peut en signaler l'utilisateur.

2.4.3 Algorithmes énigmatiques

Un outil qui regarde les design patterns de notre code ça n'existe pas, car chaque code est singulier et que ça ne peut pas être généralisé. Beaucoup de problème architecturaux au sein d'un programme sont dur à transmettre à un programme, alors que nous sommes capable de le sentir en tant que développeur il est compliqué de savoir si un programme à besoin du pattern Factory pour créer une panoplie d'objet ou bien d'un observer/observable. Ces sujets sont intéressant à étudier mais n'ont pas pu être implémenté.

Chapitre 3

Evaluation

3.1 Es-ce novateur ?

Bien qu'il existe des flopées de logiciels et plugins qui permettent d'analyser les bad smells en tout genre (repérer les duplications de code, trouver le code mort, aider à l'encapsulation...) il existe peu de logiciel capable d'analyser la structure des AntiPattern, beaucoup de résultats si nous faisons cette recherche sont reliés à de la recherche. Le fait est que pour la plupart des AntiPattern cela paraît compliqué à implémenter et à adapter à tout les programmes, ces programmes étant chacun unique.

3.2 Mode opératoire

Pour évaluer nos différents algorithmes nous utiliserons quelque projets maisons, certains qui sont petits et d'autre qui sont moyen, sachant que de gros projets existant enclenche souvent des exception du à une mauvaise utilisation de l'outil Spoon. Il y a un projet test où il n'y a quasiement rien dedans, un projet plus gros d'analyse de Bucket et un projet moyen qui remonte une Cause Effect Chain.

3.2.1 En resultat

On s'intéresse ici aux résultats apportés par les algorithmes en termes d'AntiPattern trouvés et voir si ils sont cohérents.

Pour l'algorithme de SpaghettiCode le fait qu'il cherche toute les methodes un peu partout en comparant des chaines de String fait qu'il est trop long et pose plein de

problèmes pour les grands projets. Aussi pour le projet de test 3 appels de methodes à la suite sont considérés comme spaghetti, alors que pour les autres c'est 6.

Algorithmes / resultat	ProjetTest	ProjetOPL2	ProjetOPL3	apache
Blob	1	1	3	31
SpaghettiCode	3	19	17	beaucoup trop long
TooManyParameters	1	aucun	3	176
BooleanParameters	1	0	0	21

Plus un projet est gros plus il possédera d'AntiPattern, et il est naturel de penser que les AntiPattern créent d'autres AntiPattern et code smells à leurs tours, une usine à gaz ne fait que produire de plus en plus de gaz.

3.2.2 En temps

Regardons désormais les performances en termes de temps de chaque algorithmes.

Algorithmes / temps Apache	ProjetTest	ProjetOPL2	ProjetOPL3
sans SpaghettiCode 25189	1674	2686	2629
avec SpaghttiCode indeterminé	1770	2673	2840

La performance des différents algorithmes est important car dans un gros environnement de production l'algorithme devrait traiter plusieurs centaines de classes par heure voir plus.

3.2.3 En mémoire

L'impact mémoire de notre application est faible, nous ne gardons rien en cache. Par contre nous utilisons de l'espace disque lors des clonages des différents dossiers dans notre système de modèle. La mémoire utilisée s'exprime par la relation suivante :

Taille du code de ce projet + 2*taille du projet à analyser

3.3 Voie d'amélioration

Il y a beaucoup de voies d'amélioration dans cette partie travail technique :

1. La qualité des helper, ce ne sont pour le moment que des pages Web mais l'idée de base étant de créer une interface avec le Framework Play pour interagir avec l'utilisateur et l'entrainer sur un code test à résoudre différents exercice d'AntiPattern différent.
2. Le nombre d'AntiPattern qui est très grand pour apporter un support complet au programme.
3. La qualité des AntiPattern.

Conclusion

Notre projet présente un début de réflexion et de solution au problème d'analyse d'AntiPattern et répond à notre question initiale qui était : Peut on notifier un développeur de ses AntiPattern et l'aider à les contrer ? Cependant quelques obstacles comme la qualité du résultat ou le nombre d'AntiPattern sont encore à améliorer.

Nos analyseurs et algorithmes mis en place sont variés, ils identifient différents aspects, éléments qu'un programme peut avoir et sont indépendants les un des autres. Nous proposons donc une solution qui mixe différentes techniques d'AntiPattern.

D'autres fonctionnalités pourraient être ajoutées, comme s'attaquer à d'autres types de problèmes ou bien affiner les assistances à l'utilisateur après avoir localiser ces erreurs et aussi développer de nouvelles techniques.

Annexe A

Log de l'exécution des handlers

```
TooManyParameters analyse launched.
BooleanParameter analyse launched.
Blob analyse launched.
Bucket is a blob !
Go and check this out to learn how to counter that : FightTheBlob.html
ContinuousObsolecence analyse launched.
SpaghettiCode analyse launched.
3802ms
```

Annexe B

Structure d'un page de helper

Symptome de votre AntiPattern

Votre méthode posséde plus de 3 parametres.

Pourquoi ce problème est apparu ?

Une longue liste de paramètres peut se produire après que plusieurs types d'algorithmes aient fusionnés dans une seule méthode. Une longue liste peut avoir été créée pour contrôler quel algorithme sera exécuté et comment. Les listes de paramètres longs peuvent également être le sous-produit des efforts visant à rendre les classes plus indépendantes les unes des autres. Par exemple, le code pour créer des objets spécifiques nécessaires dans une méthode a été déplacé de la méthode au code d'appel de la méthode, mais les objets créés sont transmis à la méthode en tant que paramètres. Ainsi, la classe originale ne connaît plus les relations entre les objets, et la dépendance a diminué. Mais si plusieurs de ces objets sont créés, chacun d'entre eux nécessitera son propre paramètre, ce qui signifie une liste de paramètres plus longue. Il est difficile de comprendre ces listes, qui deviennent contradictoires et difficiles à utiliser à mesure qu'ils grandissent. Au lieu d'une longue liste de paramètres, une méthode peut utiliser les données de son propre objet. Si l'objet actuel ne contient pas toutes les données nécessaires, un autre objet (qui obtiendra les données nécessaires) peut être transmis en tant que paramètre de méthode.

Traitement

- Vérifiez les valeurs transmises aux paramètres. Si certains des arguments ne sont que des résultats d'appels de méthode d'un autre objet, remplacer le paramètre par des appels de méthode. Cet objet peut être placé dans le champ de sa propre classe ou passé comme paramètre de méthode.
- Au lieu de transmettre un groupe de données reçues à partir d'un autre objet en tant que paramètres, passez l'objet lui-même à la méthode, en preservant l'entier.
- S'il existe plusieurs éléments de données non liés, vous pouvez parfois les fusionner en un seul objet de paramètre.

Résultats

- Code plus lisible et plus court.
- Ce refactoring peut aider à réveler du code dupliqué.