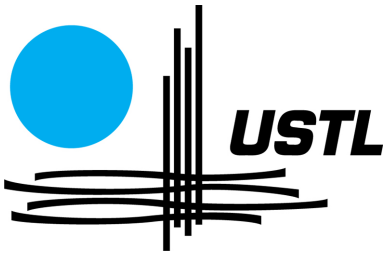

Université des Sciences et Technologies de Lille
Master Informatique 1ère année

Détection semi-automatique des patrons de mauvaise conception dans les architectures orientées objet

Projet proposé par

Alban TIBERGHIEN

encadré par

Naouel Moha
et Anne-Françoise Le Meur

Ce projet a été rendu le lundi 4 juin 2007 et soutenu le jeudi 7 juin 2007.

Table des matières

1- Introduction

Contexte

La qualité des programmes a été grandement améliorée par l'introduction des patrons de conception. Néanmoins il existe des défauts de conception qui proviennent de "mauvais" choix conceptuels et ont pour conséquence de dégrader la qualité des architectures orientées objet. Parmi ces défauts, il existe les "code-smells" et les "anti-patrons" de conception.

Les **code-smells** sont des défauts au niveau implémentation et des indicateurs ou des symptômes de la présence possible de défauts de plus haut niveau tels que les anti-patrons. Les **anti-patrons** sont des mauvaises solutions à des problèmes de conceptions récurrents qui ont un impact négatif sur la qualité de la conception d'un programme.

A ce jour et à notre connaissance, aucune étude n'a pu prouver l'impact, l'influence, des patrons de mauvaise conception sur la qualité des programmes à objets. C'est pourquoi il est aujourd'hui important de caractériser clairement l'impact des patrons de mauvaise conception sur la qualité des programmes à objets.

Problématique

Le projet qui nous a été donné s'inscrit dans le cadre de la thèse de Naouel Moha qui porte sur la détection et la correction des défauts de conception [6]. L'élaboration d'un logiciel permettant de détecter et de corriger les défauts automatiquement est un des objectifs visés par Naouel Moha. Nous intervenons dans la partie "détection des défauts de conception" de sa thèse.

Pour se faire, nous avons d'abord effectué une analyse du domaine en prenant connaissance des différents travaux de spécification des défauts qui ont été réalisés. A partir des références, nous avons proposé nos propres définitions des défauts et nous avons ensuite déterminé une méthodologie de détection manuelle. Cette méthodologie se présente sous la forme d'arbres de décision où sont définies toutes les étapes à suivre lors de l'inspection d'un code source. Cette méthodologie a ensuite été appliquée sur un programme existant de taille significative. Enfin, à partir des résultats de la détection manuelle, une comparaison entre l'outil proposé par Naouel Moha et celui proposé par Radu Marinescu a été faite dans le but de déterminer leur précision et leur rappel, qui permet d'évaluer la justesse des résultats [3].

Ce document est composé de quatre chapitres. Dans le premier chapitre, nous énumérons les définitions proposées pour les différents défauts. Dans le second chapitre, nous présentons les différents diagrammes de détection manuelle. Dans le troisième chapitre, nous faisons un bilan des défauts rencontrés dans le projet open-source analysé en donnant un exemple pour chaque défaut. Enfin, dans le dernier chapitre, nous faisons la comparaison des deux outils cités précédemment.

2- Les défauts de conception

Afin de classer un certain nombre de défauts de conception, nous avons utilisé la classification proposée par Mäntylä [7] qui permet de mieux comprendre les différents défauts et les relations qui les lient. Cette classification propose de regrouper les différents défauts par famille. Nous avons décrit chaque défaut en suivant ce patron : le (ou les) nom(s) connu(s) de ce défaut, un identifiant (ID), une description, et dans certains cas des exemples. L'identifiant servira par la suite lors du relevé des défauts.

Le principal problème rencontré pour spécifier les défauts de conception est qu'il n'existe pas de consensus sur la définition des défauts. Chacun se propose d'étudier les défauts qu'il juge intéressant en y mettant sa propre définition. Afin d'éviter toute ambiguïté, nous avons dressé la liste des défauts que nous allions traiter. Les définitions que nous leur avons attribuées sont donc une synthèse objective agrémentée de points de vue personnels.

2.1 The Bloaters

Cette famille de code-smells caractérise des entités devenues si grandes qu'elles sont difficiles à gérer.

2.1.1 Long Method

Nom(s) : Long Method [5, p. 76]

ID : `longMethod`

Description : Le principal problème d'une méthode trop longue est que le "fonctionnement" de celle-ci est difficile à définir. En effet, plus une méthode est longue, plus elle utilise de paramètres et de variables et il y a donc de fortes chances pour que cette méthode fasse plus de choses que son nom le suggère. Le fait d'éclater les méthodes trop longues en plus petites méthodes permet de mieux comprendre ce que le code fait. Cela permet donc une meilleure réutilisabilité et une meilleure flexibilité.

2.1.2 Large Class

Nom(s) : Large Class [2, p. 78]

ID : `largeClass`

Description : Ces classes sont des classes ayant trop de responsabilités. Elles ont trop d'attributs de classe et/ou trop de méthodes. Le problème est que ces classes sont trop difficiles à maintenir et à comprendre à cause de leur taille.

2.1.3 Primitive Obsession

Nom(s) : Primitive Obsession [2, p. 81]

ID : `primitiveObsession`

Description : L'utilisation abusive des primitives telles que les chaînes de caractères, les réels, et les tableaux engendre un code difficile à modifier puisque très fortement lié à ces primitives. Il est possible de définir ces primitives sous forme de classes afin de fournir un niveau d'abstraction plus élevé pour rendre le code plus clair et plus simple.

2.1.4 Long Parameter List

Nom(s) : Long Parameter List [2, p. 78]

ID : `longParameterList`

Description : Dans les langages de programmation procéduraux, l'utilisation de variables globales est considérée comme mauvaise. C'est pourquoi, dans ces langages, les fonctions ont souvent une longue liste de paramètres. L'approche objet a changé ce paradigme puisque les paramètres d'une méthode peuvent être réunis dans des objets. Ainsi, les données qu'une méthode a besoin sont directement récupérées à partir de l'objet passé en paramètre. Les listes de paramètres sont donc plus courtes et donc la méthode plus facile à utiliser et à comprendre.

2.1.5 Data Clumps

Nom(s) : Data Clumps [2, p. 81]

ID : `dataClumps`

Description : Il s'agit d'un ensemble de variables fortement liées entre elles sémantiquement. Ces variables se retrouvent constamment ensemble dans le code et peuvent être encapsulées dans des objets afin de réduire la taille des méthodes/classes.

Exemple(s) : les trois entiers pour les couleurs RGB.

2.2 The Object-Oriented Abusers

Cette famille de code-smells caractérise des cas où les choix d'implantation n'exploitent pas totalement les possibilités de la conception orientée objet.

2.2.1 Switch Statement

Nom(s) : Switch Statement [2, p. 82]

ID : `switchStatement`

Description : L'utilisation de switch/case devrait être évitée dans la programmation objet. Souvent lorsqu'un code est pensé avec des switch/case, celui-ci est dupliqué partout dans la classe. Si un cas est ajouté/enlevé, il faut mettre à jour tous les switch/case. La notion de polymorphisme est une solution plus élégante pour traiter ce problème.

2.2.2 Temporary Field

Nom(s) : Temporary Field [2, p. 84]

ID : `temporaryField`

Description : Certains objets contiennent des attributs qui ne sont pas utilisés dans tous les cas. Ces attributs ne peuvent servir, par exemple, que pendant le déroulement d'un algorithme et le reste du temps, ils ne servent à rien et donc sont vides ou contiennent des données peu pertinentes. Généralement, tous les attributs doivent être utilisés. Si seule une partie des attributs est utilisée, le code est confus et difficile à comprendre. Ce défaut est une alternative aux listes de paramètres longues. Selon Mäntylä, ce défaut apparaît lorsqu'une variable est déclarée au niveau de la classe alors qu'elle devrait l'être au niveau de la méthode. Ceci viole le principe d'encapsulation.

2.2.3 Refused Bequest

Nom(s) : Refused Bequest [2, p. 87]

ID : `refusedBequest`

Description : Ce code-smell survient quand une sous-classe utilise peu ou pas du tout un attribut/méthode qu'elle a récupéré d'un parent par héritage. Typiquement, cela signifie que la hiérarchie des classes est fausse ou mal organisée.

2.2.4 Alternative Classes with Different Interfaces

Nom(s) : Alternative Classes with Different Interfaces [2, p. 85]

ID : `alternativeClasses`

Description : Il s'agit de classes faisant la même chose mais ayant des interfaces différentes. Il s'agit d'une mauvaise utilisation de l'héritage.

2.2.5 Poor Usage of Abstract Classes

Nom(s) : Poor sage of Abstract Classes

ID : `poorUsageOfAbstractClasses`

Description : Ce défaut se produit lorsque des classes abstraites ne sont pas bien utilisées ou construites.

Exemple(s) :

- des classes abstraites sans méthodes abstraites ;
- des méthodes abstraites qui surchargent d'autres méthodes abstraites ;
- des classes abstraites sans aucune méthode ;
- des méthodes non abstraites mais non implémentées.

2.2.6 Not Enough Information Hiding

Nom(s) : Not Enough Information Hiding

ID : `notEnoughInformationHiding`

Description : Ce défaut se produit quand une classe rend visible des méthodes ou des attributs alors qu'ils auraient dû être inaccessibles. De même, de manière plus générale, lorsqu'une classe expose des informations liées à l'interface ou à la base de données. L'utilisateur dispose donc d'informations et de fonctionnalités qu'il ne devrait pas avoir.

Exemple(s) :

- l'accès à une structure de données pourrait être dangereux car l'utilisateur pourrait la manipuler comme il le souhaite et donc la corrompre ;
- les packages ou classes avec seulement (ou la plupart) des éléments publics sont susceptibles de ne pas assez encapsuler d'informations.

2.2.7 Too Much Information Hiding

Nom(s) : Too Much Information Hiding

ID : `tooMuchInformationHiding`

Description : Ce défaut se produit quand une entité ne propose pas assez de fonctionnalités pour la manipuler correctement.

Exemple(s) : une classe dispose de trop de méthodes/attributs privés.

2.2.8 Poor Usage of Interfaces

Nom(s) : Poor Usage of Interfaces

ID : `poorUsageOfInterfaces`

Description : On observe ce défaut lorsque qu'une interface est mal utilisée.

Exemple(s) :

- une classe implémente une interface déjà implémentée par une autre classe ;
- une interface définit une méthode déjà déclarée dans une interface héritée.

2.3 The Dispensables

Cette famille de code-smells caractérise des classes superflues qui pourraient être supprimées du code source.

2.3.1 Lazy Class

Nom(s) : Lazy Class [2, p. 83]

ID : `lazyClass`

Description : Ce sont des classes qui n'ont pas une grande utilité car elles ne contiennent peu ou pas de code fonctionnel. Les classes vides ou petites sont des cas spéciaux de ce type de défaut. Elles compliquent souvent la maintenance et la compréhension du projet. Dans ce cas, soit la classe est supprimée soit des fonctionnalités lui sont ajoutées.

2.3.2 Data Class

Nom(s) : Data Class [2, p. 86]

ID : `dataClass`

Description : Ce sont des classes qui ne contiennent que des attributs et leurs accesseurs. Elles n'offrent aucune fonctionnalité particulière.

2.3.3 Speculative Generality

Nom(s) : Speculative Generality [2, p. 83] / Premature Generalization / Premature Abstraction

ID : `speculativeGenerality`

Description : Ce défaut existe quand on est en présence d'un code générique ou abstrait prévu pour des usages futures. Ce code pollue inutilement le système.

2.4 The Coupler

Cette catégorie désigne les défauts qui couplent fortement des classes entre elles. Deux classes sont dites couplées si la modification de l'une demande un changement de l'autre.

2.4.1 Message Chains

Nom(s) : Message Chains [2, p. 84] / Law of Demeter

ID : `messageChains`

Description : Ce défaut désigne de longues séquences d'appels de méthodes ou de variables temporaires d'un objet à l'autre avant de pouvoir accéder à des données. Cette chaîne rend le code dépendant des relations entre un grand nombre d'objets potentiellement peu reliés.

Exemple(s) : chaînes d'invocation de méthodes tels que `A.getB().getC().do()`.

2.5 Les anti-patrons

Les anti-patrons sont des défauts de conception plus complexes. Nous avons défini une heuristique afin de caractériser les anti-patrons : un anti-patron repose sur l'existence de code-smells.

2.5.1 The Blob

Nom(s) : Blob [1, p. 73]

ID : `blob`

Description : Le Blob est une classe centralisant le traitement et donc ,par conséquent, il possède beaucoup d'attributs et de méthodes. En général, il y a peu de cohésion dans ces classes qui dépendent surtout d'autres classes où sont stockées les données. On remarque une absence de conception orientée objet (plutôt orientée procédure). Ces classes sont difficiles à réutiliser et à tester. De plus, elles utilisent beaucoup de ressources. Cet anti-patron est souvent issu de l'évolution d'un prototype ou d'un développement incrémental d'un projet. Il est difficile de faire la différence entre l'utile et le superflu.

2.5.2 Functional Decomposition

Nom(s) : Functional Decomposition [1, p. 97] / Module Mimic

ID : `functionalDecomposition`

Description : On observe cet anti-patron lorsqu'un programmeur, à l'aise avec un langage procédural, transforme ses routines en classes ignorant ainsi la conception orientée objet. Le code est complexe et il n'y a aucune hiérarchie de classes. De plus, la classe ne propose qu'une seule fonctionnalité. On peut aussi remarquer que les attributs sont "private" et que les classes ont des noms de fonction tels que 'Calculat_Interest', 'Display_Table'. Le non-respect des principes de la programmation orientée objet rend le code difficile à maintenir, à documenter, à réutiliser et à tester.

2.5.3 Spaghetti Code

Nom(s) : Spaghetti Code [1, p. 119]

ID : `spaghettiCode`

Description : C'est un système où on trouve peu de structure : peu/pas d'héritage, de réutilisation et/ou de polymorphisme. Le système inclut un petit nombre d'objets avec des méthodes trop grandes qui sont appelées une seule fois. Il y a un faible degré d'interaction entre les objets. Les méthodes n'ont pas de paramètres et utilisent des classes ou des variables globales pendant le traitement. Ces codes ne sont, en général, pas réutilisables. Le coût de maintenance de tels programmes peut être supérieur au coût de la réalisation d'une nouvelle solution.

2.5.4 Swiss Army Knife

Nom(s) : Swiss Army Knife [1, p. 197]

ID : `swissArmyKnife`

Description : Il s'agit d'une classe pour laquelle le programmeur a essayé de fournir toutes les signatures de méthodes possibles dans le but de répondre à tous les besoins. Tel un couteau suisse, beaucoup de fonctionnalités sont disponibles mais peu sont utilisées concrètement. Cet outil prend donc de la place pour rien et en devient difficile à manipuler.

Exemple(s) : les classes utilitaires.

2.6 Bilan

En l'absence de formalisme "officiel", il a été nécessaire de définir par nous même les différentes définitions des code-smells et des anti-patrons. Ce problème n'est pas rencontré, par exemple, lorsque l'étude porte sur les patrons de conception puisque le formalisme[4] proposé par Gamma et ses collaborateurs (le "Gang of Four") est considéré, par la communauté, comme la référence dans le domaine. Pour la suite du projet, nous ne considérerons plus que les définitions données précédemment. Ainsi, nous resterons cohérents tout au long de notre étude.

3- Méthodologie de détection manuelle des défauts de conception

Une fois les définitions des code-smells et des anti-patrons fixées, nous avons pu définir la méthodologie à suivre pour la détection manuelle des défauts de conception. Cette méthodologie a pour but de faciliter l'analyse des programmes en rendant systématique et sans ambiguïté le parcours des différentes classes et ainsi de laisser, à la personne chargée de l'analyse, le moins d'alternatives possibles. Le seul outil nécessaire à la recherche manuelle de défauts est l'IDE Eclipse qui met à la disposition de l'utilisateur un ensemble de fonctionnalités simples d'utilisation comme, par exemple, son explorateur de packages ou encore son outil de recherche avancé. Le fait que la méthodologie présentée ci-dessous ne nécessite qu'Eclipse sous-entend qu'aucun outil, spécialisé dans la détection de défauts, n'est utile pour effectuer de manière efficace la détection manuelle des défauts de conception. En effet nous souhaitons que cette méthodologie se base sur l'expérience de l'investigateur en tant que programmeur plutôt que sur l'utilisation d'outil d'investigation de défauts.

Les figures présentées dans ce chapitre représentent les différents diagrammes de méthodes qui permettent de détecter manuellement l'ensemble des code-smells définis précédemment.

Les diagrammes se lisent du haut vers le bas. Les rectangles indiquent de quelle manière les différentes entités logicielles vont être inspectées. Les losanges correspondent à la condition qu'il faut remplir pour descendre dans le diagramme. On descend donc dans le diagramme lorsque la condition est satisfaite. Dans le cas contraire, la détection du défaut a donc échoué et on passe au défaut suivant. Enfin, les ovales spécifient quel défaut a finalement été détecté. Nous conseillerons à l'investigateur d'appliquer l'ensemble de la méthodologie classe par classe.

3.1 Détection des code-smells

De manière générale, sur chaque diagramme, nous avons précisé sur quoi portait l'inspection. Ainsi, l'investigateur sait immédiatement sur quoi il doit se focaliser. Par exemple, pour le diagramme de la figure 3.1, il sait qu'il s'intéressera aux méthodes et aux attributs de la classe.

Sur ce même diagramme, nous pouvons remarquer la présence de valeurs seuil. En effet, lors de l'élaboration de la méthodologie, nous nous sommes rendu compte que certaines valeurs devaient être fixées pour la détection des défauts *Lazy Class* et *Large Class* de la figure 3.1. Il apparaît évident que ces valeurs ne peuvent pas être fixées arbitrairement mais à l'inverse il n'existe pas de méthodes standard pour les définir. Nous avons obtenu ces chiffres en recueillant les résultats d'analyses antérieures faites sur une dizaine de programmes. Nous avons compté les entités qui nous intéressaient dans chacun des programmes et nous avons appliqué des algorithmes de clustering pour extraire trois clusters correspondant aux trois catégories suivantes : petit, moyen et grand. L'analyse de ces données nous a permis de récupérer les valeurs seuil associées à chacune des catégories.

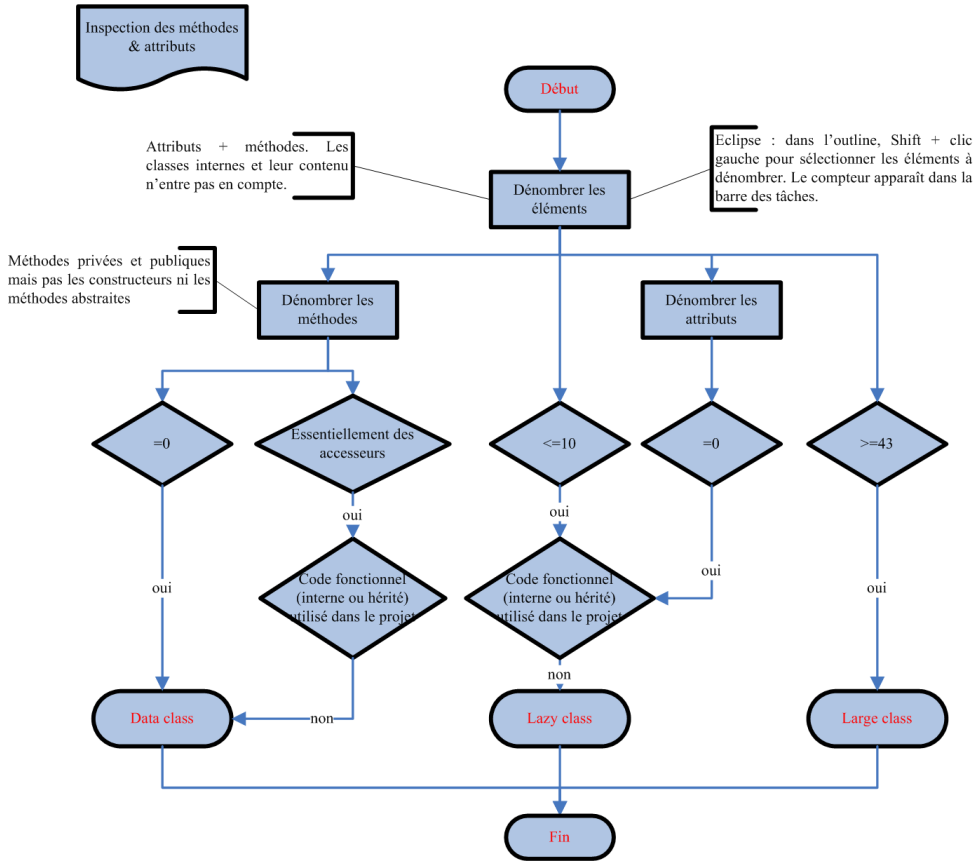

FIG. 3.1 – Diagramme permettant de détecter les défauts : Data Class, Large Class et Lazy Class

Dans le diagramme de la figure 3.2, nous nous intéressons à la visibilité des attributs/méthodes. Nous avons étudié les différentes situations dans lesquelles un attribut public ou une méthode publique pouvait intervenir et nous avons déterminé lesquelles étaient des défauts. Nous avons distingué trois types d'utilisation d'un attribut public :

- un attribut utilisé uniquement dans la classe courante : problème car il est consultable et modifiable de l'extérieur. Il aurait dû être privé.
- un attribut utilisé dans la classe courante et à l'extérieur mais seulement en lecture ou en écriture : problème car la consultation ou la modification reste possible. Il aurait dû être privé et disposer d'un getter ou d'un setter.
- un attribut utilisé dans la classe courante et à l'extérieur en lecture et écriture : aucun problème. Le programmeur a juste voulu faire l'économie d'un getter et d'un setter.

Nous avons appliqué le même raisonnement pour les méthodes publiques. Nous avons distingué deux types d'utilisation :

- une méthode utilisée uniquement dans la classe courante : problème car soit cela dévoile l'implémentation de la classe, soit cela modifie l'instance. La méthode aurait dû être privée.
- une méthode utilisée dans la classe courante et à l'extérieur : aucun problème.

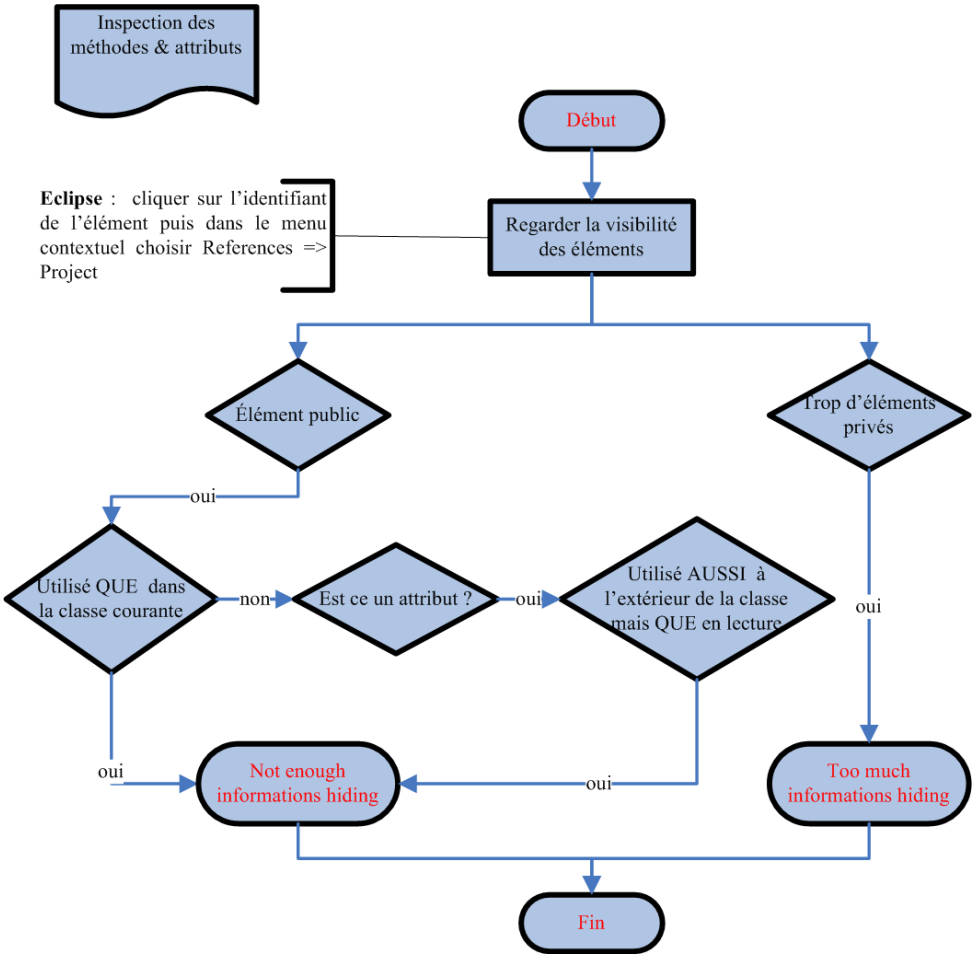

FIG. 3.2 – Diagramme permettant de détecter les défauts : Not enough information hiding et Too much information hiding

Sur la figure 3.3, il faut remarquer que la solution d'un défaut de type "Data Clumps" est elle-même un *Data Clumps* mais qui n'est plus considéré comme un défaut. Lors de la détection manuelle, il faut donc faire attention à la nature du *Data Clumps* : soit il s'agit du défaut soit il s'agit de sa solution. Ce que nous entendons par "Nom générique", c'est un nom de méthode issu du monde de la programmation orientée processus : `run`, `execute`, `initialize`, `compute`, etc.

Comme sur le diagramme de la figure 3.1, nous avons défini des valeurs seuil pour la détection des défauts *Long Parameter List* et *Long Method* de la figure 3.4. Ces valeurs seuil ont été déterminées de la même manière que pour la première méthodologie. Pour la détection du défaut *Switch Statement*, savoir si la "modification de la structure de contrôle est répercutée ailleurs dans le code" consiste à vérifier si l'objet sur lequel on fait le test prend une nouvelle valeur. Dans ce cas, cela impliquera un ajout de "if" ou de "case" dans la ou les stuctures de contrôle relatives à cet objet. Ensuite pour savoir si "le traitement aurait pu être géré avec le polymorphisme", il faut se demander si le traitement aurait pu être déporté dans les objets impliqués.

Le diagramme de la figure 3.5 traite de l'héritage. De manière générale, on constate qu'une interface doit être implémentée par une et une seule classe (abstraite à priori) et que cette classe doit être héritée par plusieurs classes filles.

Le diagramme de la figure 3.6 traite des classes abstraites.

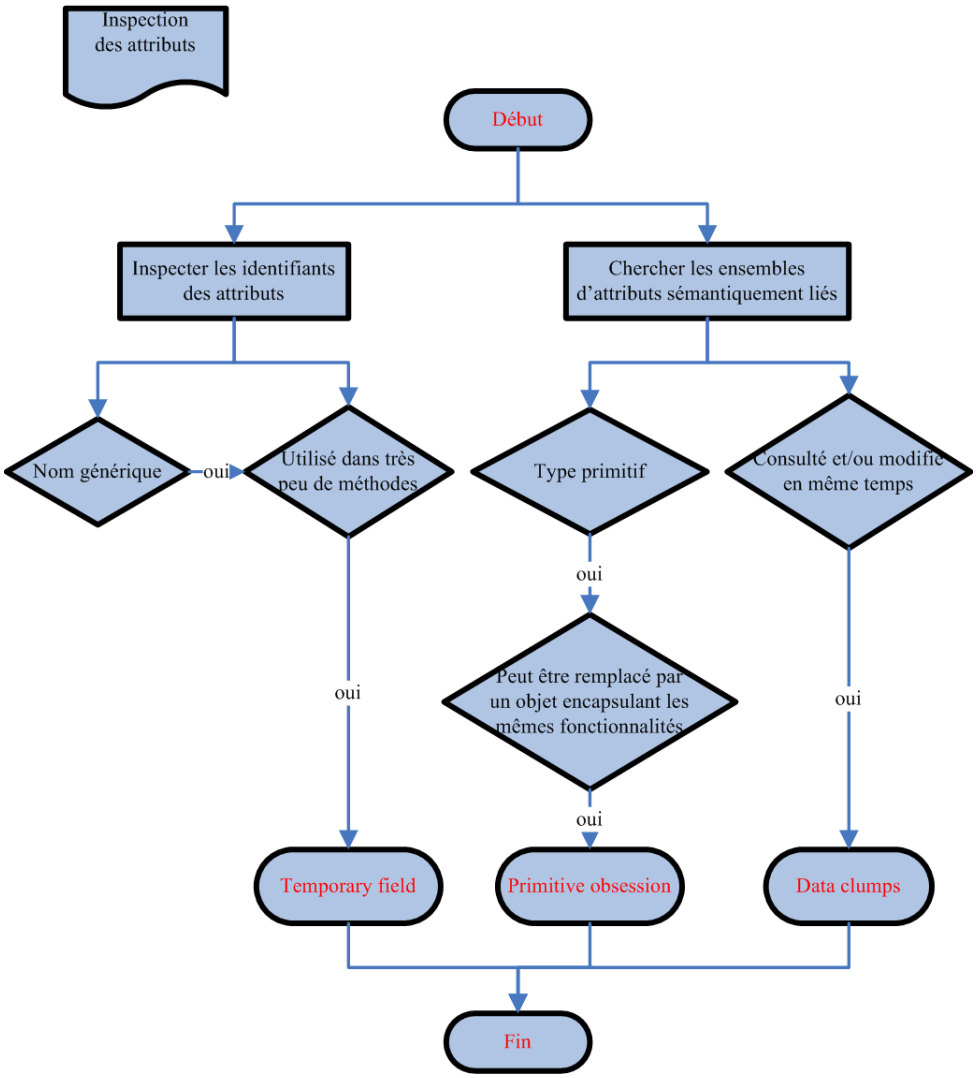

FIG. 3.3 – Diagramme permettant de détecter les défauts : Temporary Field, Primitive Obsession et Data Clumps

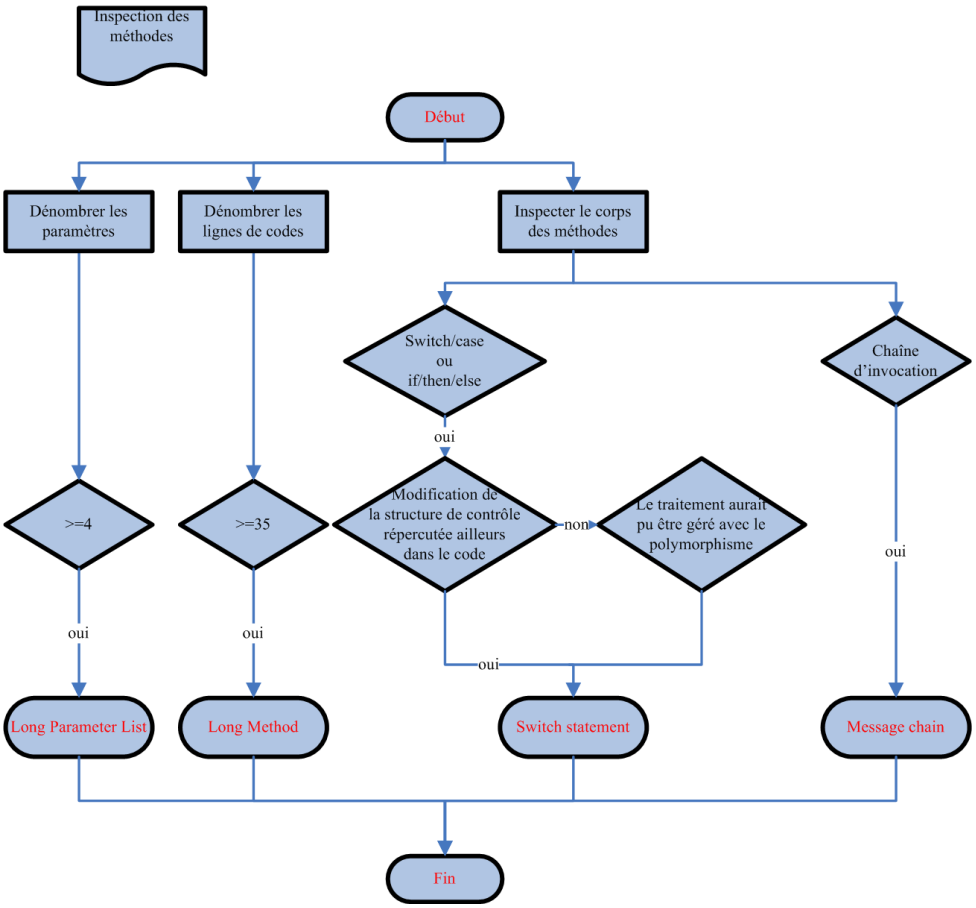

FIG. 3.4 – Diagramme permettant de détecter les défauts : Long Parameter List, Long Method, Switch Statement et Message Chain

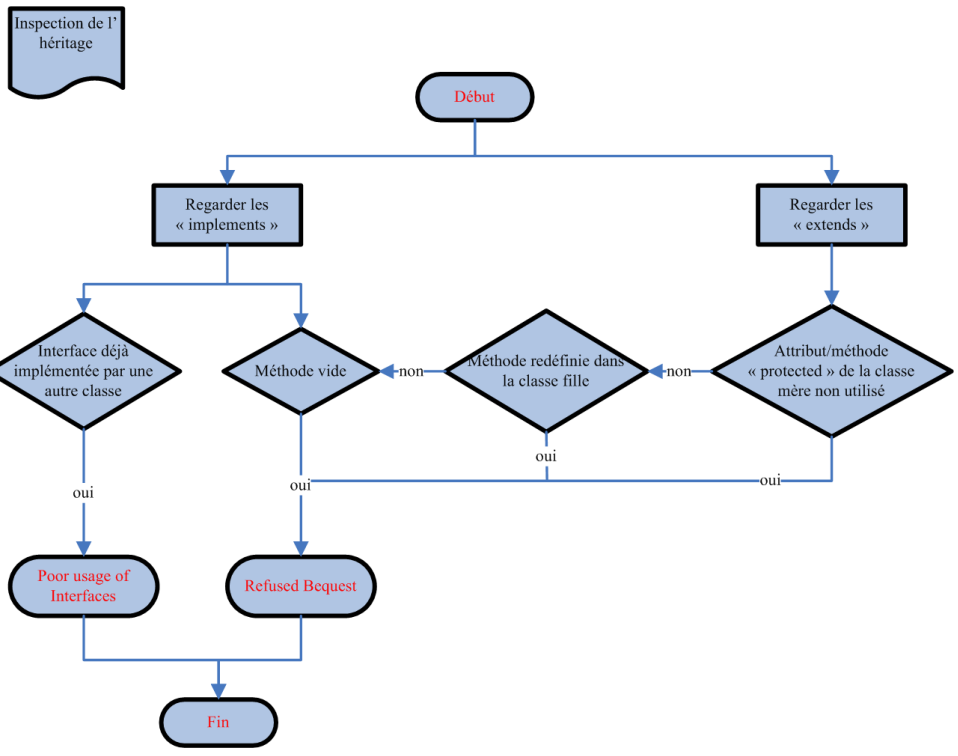

FIG. 3.5 – Diagramme permettant de détecter les défauts : Poor usage of interfaces et Refused Bequest

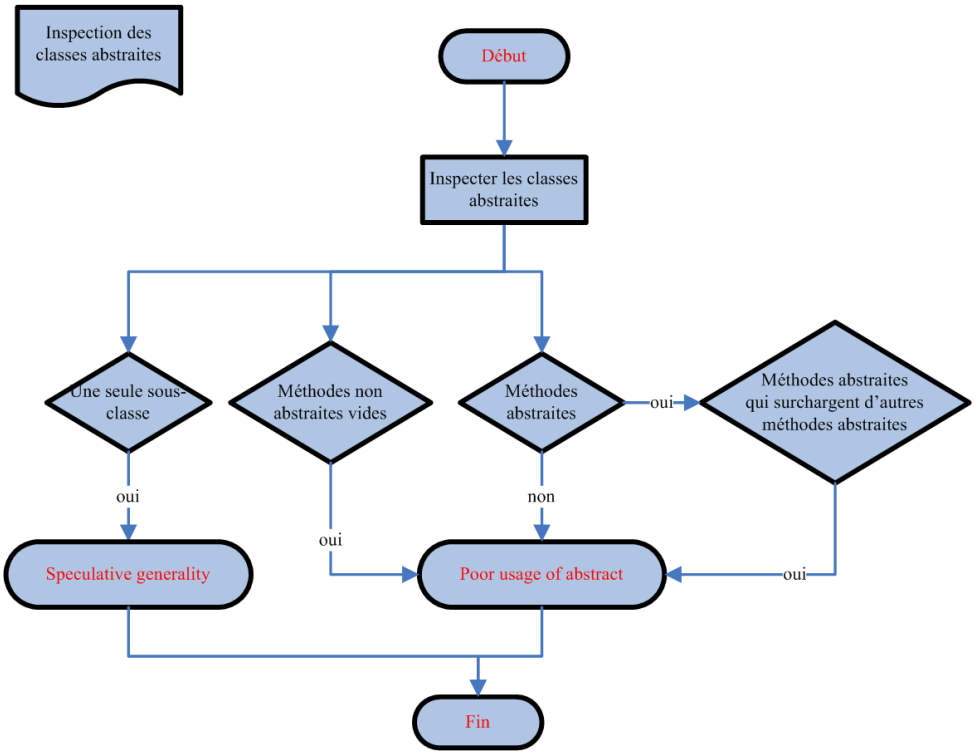

FIG. 3.6 – Diagramme permettant de détecter les défauts : Speculative Generality et Poor usage of abstract

3.2 Détection des anti-patrons

Le principe de la détection des anti-patrons est que l'investigateur ne reparcourt plus toutes les classes du système mais qu'il puisse déduire, des code-smells détectés précédemment, la présence d'anti-patrons. Comme dit plus haut, nous avons défini les anti-patrons comme étant basés sur l'existence de code-smells. Les diagrammes de la figure 3.7 et de la figure 3.8 ont donc été définis conformément à cette heuristique.

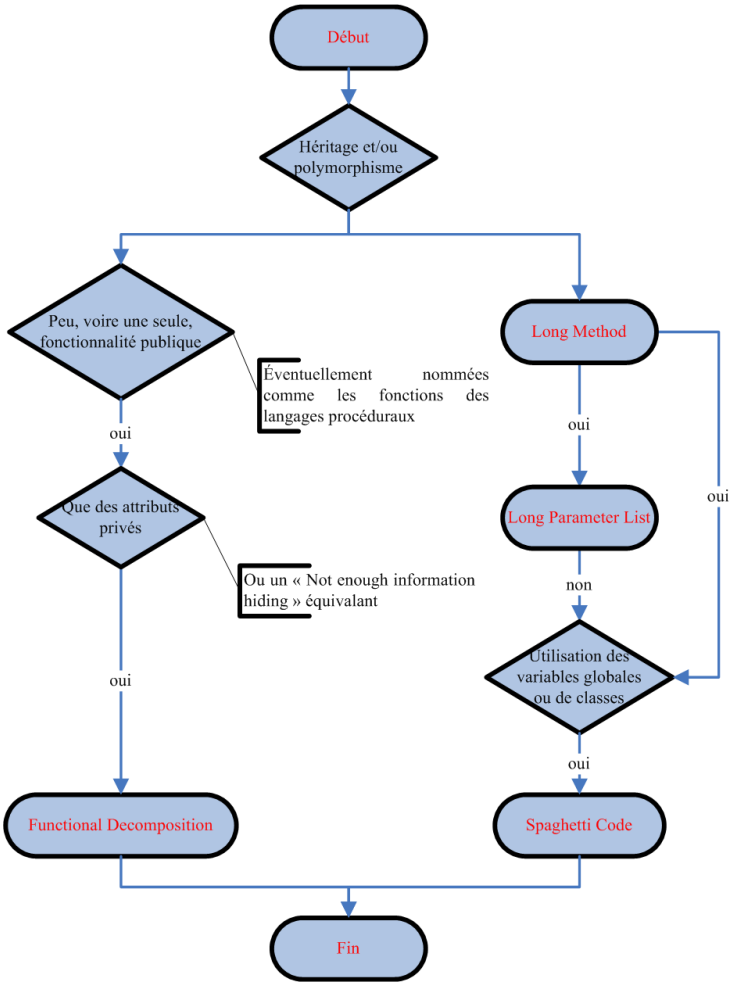

FIG. 3.7 – Diagramme permettant de détecter les défauts : Functionnal Decomposition et Spaguetti Code

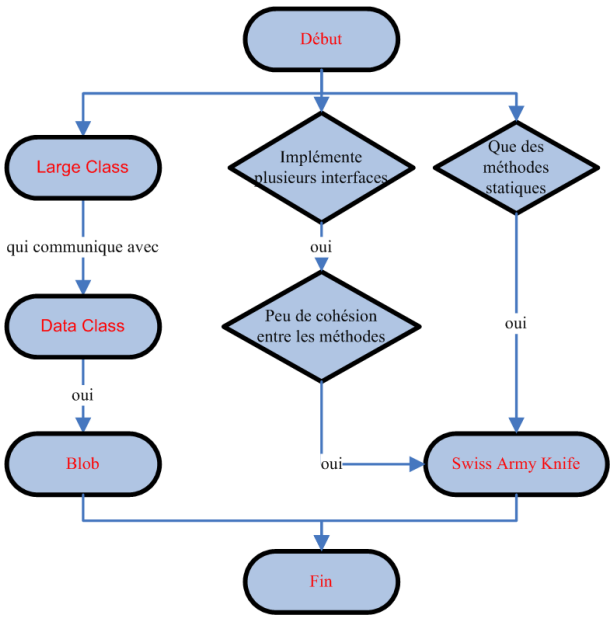

FIG. 3.8 – Diagramme permettant de détecter les défauts : Blob et Swiss Army Knife

3.3 Bilan

A partir des définitions proposées dans le premier chapitre, nous avons pu proposer une méthodologie simple et efficace pour la détection manuelle. Plusieurs itérations ont été faites sur cette méthodologie. Dans un premier temps, seules les définitions ont servi. Ensuite la méthodologie a été appliquée plusieurs fois sur un des packages de l'application que nous devions analyser. A chaque itération, la méthodologie est devenue meilleure en efficacité mais en simplicité aussi.

4- Analyse et recherche des défauts de conception dans GanttProject

Précédemment nous vous avons présenté la méthodologie que nous avions élaborée. Cette méthodologie a été faite pour fonctionner sur tous types architecture logicielle orientée objet. En effet, la méthodologie n'a pas été conçu pour fonctionner uniquement avec GanttProject. Nous ne savions pas à l'avance quels seraient les défauts détectés.

Les résultats de cette détection manuelle nous serviront de témoin lors de l'expérimentation que nous ferons par la suite pour comparer les outils de détections automatique.

4.1 Présentation du logiciel à analyser

Pour notre analyse nous utiliserons la version 1.10.2 de GanttProject développé en Java.

4.1.1 Le projet GanttProject

GanttProject est un outil de gestion de projet. Ce projet a été initié par des élèves de l'Université de Marne La Vallée en 2003 et est maintenant proposé sous licence libre (GPL). GanttProject permet la planification de projets à l'aide de diagrammes de Gantt. Il propose les fonctionnalités de base de ce type de diagrammes, comme la création des tâches, l'affectation des ressources, la gestion des dépendances et de l'avancement.

GanttProject est un projet composé de 27 packages et sous packages, environ 240 classes et près de 22000 lignes de code. Les 15 packages principaux appartiennent tous au package `net.sourceforge.ganttproject` :

- `action` : actions à faire suite à une interaction de l'utilisateur.
- `chart` : implémentation des diagrammes.
- `document` : permet l'accès aux fichiers externes soit sur le système de fichier local soit sur le réseau.
- `export` : permet d'exporter les données des diagrammes dans d'autres formats.
- `filter` : filtres permettant de n'importer que les types de fichiers reconnus par GanttProject.
- `gui` : implémentation de l'interface graphique.
- `io` : gestion des entrées/sorties (i.e. ouverture/fermeture/sauvegarder) du projet sous différents formats.
- `langage` : permet de gérer la langue du logiciel.
- `parser` : permet de parser les fichiers XML.
- `resource` : permet de modéliser un projet.
- `role` : permet de définir les différents rôles que peut avoir une personne assignée à une tâche (ex : développeur, chef de projet ...).
- `shape` : permet de définir les différentes formes utilisées pour le diagramme.
- `task` : permet de modéliser les tâches.
- `time` : permet la gestion du temps et du calendrier.
- `util` : fournit divers outils indépendants du projet en lui-même.

Nous avons jugé que GanttProject était un bon sujet d'étude car il s'agit d'un projet issu du monde open-source à la base développé par des étudiants n'ayant, a priori, pas beaucoup d'expérience dans le développement logiciel. Comme ce projet a connu un certain succès, il est vite devenu un travail collaboratif. Il y a ainsi plusieurs contributeurs/programmeurs et donc un risque d'avoir des défauts. De plus, certaines parties du projet ont été réalisées grâce à des outils de génération automatique de code.

4.2 Défauts de conception détectés

La détection manuelle des défauts a été faite en une vingtaine d'heure. Nous allons vous présenter dans cette section les résultats de la détection dans un premier temps et un exemple significatif pour chaque défaut dans un second temps.

4.2.1 Résultat de l'analyse

Nous avons reporté les résultats de l'analyse dans le tableau 4.1 et le tableau 4.2. Nous avons classé les défauts non plus par famille mais par ordre décroissant de défauts détectés.

Nom du code-smell	Nombre de défauts détectés dans GanttProject
Lazy Class	63 (dont 24 dans des classes internes)
Data Class	75 (dont 23 dans des classes internes)
Long Parameter List	54 (dont 10 dans des classes internes)
Not enough information hiding	54 (dont 7 dans des classes internes)
Long Method	45 (dont 4 dans des classes internes)
Refused Bequest	23 (dont 4 dans des classes internes)
Temporary Field	18
Switch Statement	11 (dont 4 dans des classes internes)
Poor Usage of Interfaces	10
Data Clumps	10 (dont 1 dans une classe interne)
Large Class	9
Speculative Generality	8
Message Chains	8
Aternative Classes With Different Interfaces	7 (dont 2 dans des classes internes)
Poor Usage of Abstract Classes	7
Primitive Obsession	2
Too much information hiding	2
406 code-smells détectés manuellement au total (dont 79 dans des classes internes)	

TAB. 4.1: Résultats de la recherche des code-smells

Nom de l'anti-patron	Nombre de défauts détectés dans GanttProject
Functional Decomposition	18
Spaghetti Code	12
Swiss Army Knife	8
Blob	8
46 anti-patrons détectés manuellement au total	

TAB. 4.2: Résultats de la recherche des anti-patrons

4.2.2 Exemple de défauts détectés dans GanttProject

Tous les code-smells et les anti-patrons détectés ont été répertoriés dans deux fichiers XML séparés : codesmells_gantt.xml et antipatterns_gantt.xml. Le premier faisant 4140 lignes et le second 446 lignes, il n'était pas envisageable de les fournir avec ce document. En revanche, les DTDs de ces deux fichiers sont disponibles en AnnexeA et en AnnexeB. Afin d'illustrer les différents défauts présentés dans le premier chapitre, voici, pour chacun d'entre eux, un exemple représentatif rencontré dans GanttProject :

Long Method

Localisation : net.sourceforge.ganttproject.GanttGraphicArea
Justification : Le constructeur de cette classe est composé de 480 lignes de code.

Large Class

Localisation : net.sourceforge.ganttproject.ResourceLoadGraphicArea
Justification : Cette classe est constituée de 44 attributs et des 29 méthodes.

Primitive Obsession

Localisation : net.sourceforge.ganttproject.action.DeleteHumanAction
Justification : L'attribut `myWebLink` de type String est utilisé pour représenter une URL.

Long Parameter List

Localisation : net.sourceforge.ganttproject.export.ProjectExportData
Justification : Cette classe dispose d'un constructeur ayant 10 paramètres.

Data Clumps

Localisation : net.sourceforge.ganttproject.GanttPrintable
Justification : Les attributs entiers x et y (resp. abscisse et ordonnée d'un point) sont consultés et modifiés en même temps dans le code.

Switch Statement

Localisation : net.sourceforge.ganttproject.GanttProject
Justification : Un "switch" est fait sur les différents types de documents que GanttProject peut gérer. Si un nouveau type de document venait à être géré, il faudrait modifier le code-source pour pouvoir rajouter le cas dans le "switch".

Temporary Field

Localisation : net.sourceforge.ganttproject.task.algorithm.RecalculateTaskScheduleAlgorithm
Justification : L'attribut privé `myEntranceCounter` n'est utilisé que dans la méthode `run(..)`.

Refused Bequest

Localisation : net.sourceforge.ganttproject.gui.about.AboutAuthorPanel
Justification : Cette classe hérite de la classe abstraite `net.sourceforge.ganttproject.gui-.options.GeneralOptionPanel` qui fournit une méthode abstraite `initialize()`. Celle-ci n'a pas été implémentée dans la classe fille.

Poor Usage of Abstract Classes

Localisation : net.sourceforge.ganttproject.gui.projectwizard.WizardImpl
Justification : Cette classe abstraite ne propose aucune méthode abstraite.

Not Enough Information Hiding

Localisation : net.sourceforge.ganttproject.GanttExportSettings
Justification : Des attributs ont été déclarés "public" alors qu'ils ne sont utilisés que dans la classe. Rien n'empêche une éventuelle consultation ou modification de ces attributs à l'extérieur de la classe.

Too Much Information Hiding

Localisation : net.sourceforge.ganttproject.gui.GanttDialogPerson
Justification : Cette classe est composée de 29 éléments privés pour seulement un constructeur et une méthode publique.

Poor Usage of Interfaces

Localisation : net.sourceforge.ganttproject.gui.projectwizard.RoleSetPage
Justification : L'interface `net.sourceforge.ganttproject.gui.projectwizard.Wizard.Page` est déjà implémentée par `net.sourceforge.ganttproject.gui.projectwizard.ProjectNamePage`.

Lazy Class

Localisation : net.sourceforge.ganttproject.parser.ParsingContext
Justification : Cette classe est composée de trois éléments : un attribut entier, le getter et le setter. Cette classe ne dispose d'aucun code fonctionnel.

Data Class

Localisation : net.sourceforge.ganttproject.gui.UIConfiguration
Justification : Cette classe dispose de six attributs, un constructeur public et de 10 getters/setters. De plus, cette classe ne dispose d'aucun code fonctionnel.

Speculative Generality

Localisation : net.sourceforge.ganttproject.export.ProjectExportProcessor
Justification : Cette classe abstraite n'est héritée qu'une seule fois.

Message Chains

Localisation : net.sourceforge.ganttproject.GanttTree
Justification : Dans la méthode cutSelectedNode() :

```
1 AdjustTaskBoundsAlgorithm alg = getTaskManager().getAlgorithmCollection()
2                                 .getAdjustTaskBoundsAlgorithm();
3 alg.run(taskFather);
```

Comme on peut le voir dans cette exemple, trois appels de getters sont faits pour récupérer l'objet voulu et ensuite seulement l'action est invoquée sur l'objet. Cette chaîne d'invocation lie fortement les trois classes impliquées.

The Blob

Localisation : net.sourceforge.ganttproject.GanttGraphicArea

Justification : Cette classe est une large classe composée de 81 éléments. Elle communique avec trois Data Classes : `net.sourceforge.ganttproject.GanttGraphicArea#GanttPaintParam`, `net.sourceforge.ganttproject.GanttTask` et `net.sourceforge.ganttproject.gui-.UIConfiguration`.

Functional Decomposition

Localisation : net.sourceforge.ganttproject.time.gregorian.GregorianTimeUnitStack

Justification : Cette classe est composée de huit attributs privés et de cinq attributs publics utilisés que dans cette classe. Elle ne fournit que le méthode `createTimeFrame`.

Spaghetti Code

Localisation : net.sourceforge.ganttproject.GanttGraphicArea

Justification : 10 méthodes de cette classe sont des Long Methods qui utilisent les attributs de la classe.

Swiss Army Knife

Localisation : net.sourceforge.ganttproject.util.ColorConvertion

Justification : Cette classe n'est composée que de deux méthodes statiques indépendantes l'une de l'autre.

4.3 Bilan

La détection manuelle nous a permis de lister de manière exhaustive les défauts de conception dans GanttProject. Cette liste de défauts constitue pour nous le témoin de l'expérimentation suivante qui confrontera deux outils de détection automatique de défauts. En effet, nous supposerons ici que l'investigateur n'a oublié aucun défaut.

5- Comparaison des outils de détection automatique de défauts de conception

L'analyse manuelle de GanttProject a permis de recenser tous les code-smells et les anti-patrons présents dans le programme. Cette analyse qui sert de témoin va permettre d'évaluer les outils de détection automatique. Les deux outils à confronter sont Ptidej, auquel Naouel Moha contribue, et iPlasma proposé par Radu Marinescu. Avant de commencer l'analyse, il nous a fallu prendre en main les outils.

5.1 Présentation des outils

DECOR est un outil qui permet la spécification et la détection de défauts tels que les code-smells et les anti-patrons de conception en utilisant un vocabulaire standard et un langage dédié. Ptidej regroupe une suite logicielle permettant d'évaluer et d'améliorer la qualité des programmes orientés objets, de faire de la rétro-conception à partir de langage comme AOL, C/C++ ou encore Java et de promouvoir les patrons de conception. Ptidej intègre DECOR et fournit des algorithmes de visualisation pour faciliter la compréhension des défauts de conception.
`http ://www-etud.iro.umontreal.ca/~mohanaou/DECOR/DECOR.htm`.

iPlasma est un environnement intégré pour l'analyse de la qualité des logiciels basés sur des technologies à objets. Cela inclut un support pour toutes les phases nécessaires à l'analyse : de l'extraction de modèle (incluant le parsing de fichiers C++, Java and C#) jusqu'à l'analyse basée sur des métriques de haut niveau ou encore la détection de duplication de code. IPlasma possède trois principaux avantages : l'extensibilité des analyses supportées, l'intégration avec d'autres outils d'analyse et l'extensibilité, comme utilisé dans le passé pour analyser des projets comportant des millions de lignes de code.
`http ://loose.upt.ro/iplasma/`.

5.2 Utilisation de iPlasma

La détection de défauts se fait par raffinement successifs grâce à des regroupements et/ou à des filtres. Le principe est de partir d'un corpus de classes et de lui appliquer successivement les algorithmes de regroupement et de filtrage afin d'obtenir un sous-ensemble souhaité. A partir du corpus initial, il faut forcément faire un premier regroupement et seulement ensuite il est possible d'alterner les algorithmes de regroupement et/ou de filtrage. Le menu "Property" permet d'afficher diverses informations numériques relatives au sous-ensemble. Il est aussi possible de définir ses propres filtres. Ceux-ci sont des prédicats basés sur les différentes métriques.

5.2.1 Algorithmes de regroupement

Afin d'utiliser au mieux IPlasma, nous avons proposé une classification des différents algorithmes de regroupement. Bien que cette classification ne fasse pas partie de notre étude, il faut noter que celle-ci a été établie de manière relativement intuitive. Lorsque le nom du regroupement n'était pas assez explicite, nous avons donné une définition induite empiriquement. En effet, nous avons déduit la fonction des algorithmes de regroupement, des propriétés et des filtres en les appliquant à GanttProject et en identifiant la nature des entités affichées. Ces définitions sont donc liées au logiciel analysé.

Nous pouvons distinguer six catégories :

1. **les anti-patrons de Marinescu** : *Identify Disharmony* (God Class, Feature Envy, Data Class, Brain Method et Brain Class), *Classification Disharmony* (Refused Parent Bequest,Tradition Breaker et Futile Hierarchy), *Collaboration Disharmony* (Shotgun Surgery, Intensive Coupling et Extensive Coupling) et *Identify Harmony* (criticals methods).
2. **les regroupements par entités** : *attribute group*, *class group*, *global variable group*, *local variable group*, *method group*, *namespace group*, *package group*, *global function group*, *parameter group*, *group of changing classes*, *group of changing method* et *group of variables of this type*.
3. **les regroupements liés à l'héritage** : *all descendants*, *current class and ancestors*, *methods inherited but not overridden*, *methods not inherited*, *methods inherited but not overridding*, *methods inherited but overridden*, *methods overridden*, *overwriting methods* et *refused ancestors*.
4. **les regroupements liés à la structure du système** : *accessed model data*, *operations calling me*, *methods accessing variables*, *operations called*, *variables accessed heavy called methods*, *heavy service providers*, *base classes*, *derived classes*, *foreign data* et *type of variables*.
5. **les regroupements liés à la cohésion entre les objets** : *Data Providers For Large-God Classes*, *External Service Providers (methods)*, *external service providers for class*, *foreign data providers* et *subclasses dependencies*
6. **les regroupements liés à la duplication de code** : *same hierarchy duplicated methods*, *same hierarchie class duplication* et *unrelated methods with duplication*.

5.3 Utilisation de Ptidej

L'utilisation de Ptidej est relativement intuitive. Deux fonctionnalités nous intéressent pour la détection des défauts : les onglets "Code Smells" et "Anti Patterns". La première fonctionnalité propose un ensemble de boutons permettant de lancer la détection des différents code-smells tandis que la seconde propose un ensemble de cases à cocher pour définir les anti-patrons à rechercher. La manipulation du modèle de classe se fait via un diagramme UML. L'identification des défauts est donc représenté par un carré rouge au dessus de la classe. Cette visualisation des défauts est persistante, ce qui permet de cumuler les défauts. Lorsque l'interface graphique ne permet pas de détecter le défaut souhaité, il existe deux alternatives pour créer ses propres règles de détection : en utilisant l'éditeur intégré ou en codant directement la règle grâce à l'API que Ptidej propose.

5.3.1 Spécification de nouvelles règles de détection

L'éditeur intégré de Ptidej permet de définir textuellement des ensembles de règles de détection (ou "rule card"), et donc les règles elles-mêmes. Une grammaire BNF (*Backus-Naur Form*) qui permet de déterminer la syntaxe exacte d'un langage (cf. AnnexeC) a été utilisée pour formaliser l'écriture des règles. Ces "rule cards" décrivent un défaut de conception, les code smells sur lesquels il repose et/ou les relations qu'il existe entre ces code-smells.

Dans l'API, les règles sont codées directement en Java en utilisant un méta-modèle qui fournit tous les constituants des programmes (classes, méthodes, variables, associations, etc). Nous n'avons pas utilisé l'API pour définir de nouvelles règles car iPlasma ne permet de faire ce genre de chose.

5.4 Calcul du la précision et du rappel

5.4.1 Définitions

La précision permet d'évaluer le nombre de vrai défauts détectés et le rappel d'évaluer le nombre de vrai défauts oubliés par les algorithmes. La précision mesure le degré auquel un outil de détection fournit des résultats corrects alors que le rappel mesure si l'outil est à même de détecter tous les défauts existants.

$$\text{Précision} = \frac{|\{\text{défauts détectés manuellement}\} \cap \{\text{défauts détectés par l'outil}\}|}{|\{\text{défauts détectés par l'outil}\}|}$$

$$\text{Rappel} = \frac{|\{\text{défauts détectés manuellement}\} \cap \{\text{défauts détectés par l'outil}\}|}{|\{\text{défauts détectés manuellement}\}|}$$

5.4.2 Utilisation des outils (Méthodes de détection)

Avant de commencer la détection, nous avons dû comprendre comment les outils fonctionnaient et sur quelles notions ils se basaient. Les deux outils ont une notion pré-requise en commun : les métriques. Une métrique est un moyen permettant de connaître la distance entre deux points. Appliquée à la production logicielle, une métrique est un indicateur d'avancement ou de qualité des développements logiciels. Comme ce projet ne porte pas sur l'utilisation des métriques dans la détection des défauts de conception, nous ne détaillerons pas toutes les métriques disponibles dans les deux outils mais seulement celles que nous avons utilisées pour spécifier les règles de détection automatique :

- LOC (Line Of Code) : nombre de lignes de code dans une méthode ou une classe.
- NOP (Number Of Parameter) : nombre de paramètres d'une méthode.
- NAbsM (Number of Abstract Method) : nombre de méthodes abstraites dans une classe.
- NOD (Number Of Descendant) : nombre d'objets qui héritent ou implémentent l'objet courant
- NOA (Number Of Attributs) : nombre d'attributs dans la classe
- NOM (Number Of Attributs) : nombre de méthodes dans la classe.

Nous avons répertorié dans le tableau 5.1 les différentes méthodes à suivre pour détecter chaque défaut. Le symbole "ensemble vide" ($\varnothing$) signifie que l'outil ne permet pas de détecter le défaut. Pour les règles d'iPlasma, le mot-clé *group* symbolise le sous-menu Group de l'interface graphique de iPlasma et le mot-clé *filter* symbolise le sous-menu Filter. Lorsque le mot-clé *filter* est suivi d'un astérisque, cela signifie que le filtre n'est pas implémenté par défaut et qu'il faut le définir soi-même. Pour les règles de Ptidej, le mot-clé "rule card" signifie que le défaut a été détecté grâce à une règle définie par nos soins. Le mot-clé "API" signifie qu'il est possible de détecter le défaut en implémentant directement la règle avec l'API Ptidej.

Toutes les méthodes de détection de iPlasma sont composées du filtre "not inner class". Nous avons dû systématiquement rajouter ce filtre car Ptidej ne détecte pas les défauts dans les classes internes donc nous n'avons pas tenu compte des classes internes lors de la détection avec iPlasma pour ne pas fausser les résultats.

Pour certains défauts, nous avons rajouté la mention "Host" qui signifie que ce sont les classes contenant au moins un défaut qui seront comptabilisées et non les défauts eux-mêmes. Cela est dû au fait que lors de la détection manuelle, les différents défauts ont été identifiés par le nom de la classe qui les contenait.

Défauts	Méthodes de détection	
	iPlasma	PtiDej
The Bloaters		
Long Method (Host) (méthode par défaut)	group “class group” + filter “model class” + filter “not inner class” + filter* “Brain Method is true”	onglet “Code Smells” + bouton “LongMethodDetection”
Long Method (Host) (méthode personnelle)	group “class group” + filter “model class” + filter* “not inner class” + filter* “LOC >= 35”	rule card (cf.ÃnnexeD)
Large Class (méthode par défaut)	group “class group” + filter “class model” + filter* “not inner class” + filter* “Brain Class”	onglet “Code Smells” + bouton “LargeClassDetection”
Large Class (méthode personnelle)	∅	rule card (cf.ÃnnexeE)
Primitive Obsession	∅	∅
Long Parameter List (Host)	group “class group” + filter “model class” + filter* “not inner class” + filter* “NOP >= 4”	API
Data Clumps	∅	∅
The Object-Oriented Abusers		
Switch Statement	∅	∅
Temporary Field (Host)	group “class group” + filter “class model” + filter* “not inner class” + filter “Temporary Field Host” + filter* “not Data Class”	API
Refused Bequest (algo 1)	group “class group” + filter “class model” + filter*“not inner class” + filter “Refused Parent Bequest”	∅
Refused Bequest (algo 2)	group “class group” + filter “class model” + filter* “not inner class” + filter “Refused Parent Bequest 2”	∅
Alternative Classes with Different Interfaces	∅	API
Poor Usage of Abstract Classes	group “class group” + filter “class model” + filter* “not inner class” + filter* “not is interface” + filter “is abstract” + filter* “NAbsM==0”	API
The Object-Oriented Abusers (suite)		
Not Enough Information Hiding	∅	API
Too Much Information Hiding	∅	API
Poor Usage of Interfaces	group “class group” + filter “class model” + filter* “not inner class” + filter “is interface” + filter* “NOD>1”	API
The Dispensables		
Lazy Class	∅	API
Data Class	group “class group” + filter “class model” + filter* “not inner class” + filter “Data Class”	onglet “Code Smells” + bouton “DataClass”
Speculative Generality	group “class group” + filter “class model” + filter* “not inner class” + filter* “not is interface” + filter “is abstract” + filter* “NOD==1”	∅
The Coupler		
Message Chains	∅	∅

Les Anti-patrons		
The Blob	group “class group” + filter “class model” + filter* “not inner class” + filter “God Class”	onglet “Anti Patterns” + checkbox “Blob”
Functional Decomposition	∅	onglet “Anti Patterns” + checkbox “Functional Decomposition”
Spaghetti Code	∅	onglet “Anti Patterns” + checkbox “SpaghettiCode”
Swiss Army Knife	∅	onglet “Anti Patterns” + checkbox “SwissArmyKnife”

TAB. 5.1: Méthode de détection des défauts pour Ptidej et iPlasma

5.4.3 Résultats

Les tableaux 5.2 et 5.3 regroupent les résultats de la précision et du rappel pour chaque défaut détecté par les outils.

		IPlasma			PtiDej		
Nom du code-smell	**Réels défauts**	Défauts détectés	Précision	Rappel	Défauts détectés	Précision	Rappel
The Bloaters							
Long Method (Host) (méthode par défaut)	41	11	11/11 (100%)	11/41 (26,82%)	12	11/12 (91,67%)	11/41 (26,82%)
Long Method (Host) (méthode personnelle)	41	49	39/49 (79,59%)	39/41 (95,12%)	94	36/94 (38,3%)	36/41 (87,8%)
Large Class (méthode par défaut)	9	0	0/0 (0%)	0/9 (0%)	0	0/0 (0%)	0/9 (0%)
Large Class (méthode personnelle)	9	∅			17	9/17 (52,94%)	9/9 (100%)
Primitive Obsession	2	∅			∅		
Long Parameter List (Host)	44	44	44/44 (100%)	44/44 (100%)	∅		
Data Clumps	9	∅			∅		
The Object-Oriented Abusers							
Switch Statement	7	∅			∅		
Temporary Field (Host)	18	73	13/73 (17,8%)	13/18 (72,22%)	∅		
Refused Bequest (algo 1)	19	0	0/0 (0%)	0/19 (0%)	∅		
Refused Bequest (algo 2)	19	1	0/1 (0%)	0/19 (0%)	∅		
Alternative Classes with Different Interfaces	5	∅			∅		
Poor Usage of Abstract Classes	7	7	7/7 (100%)	7/7 (100%)	∅		
Not Enough Information Hiding	47	∅			∅		
Too Much Information Hiding	2	∅			∅		
Poor Usage of Interfaces	10	20	10/20 (50%)	10/10 (100%)	∅		
The Dispensables							
Lazy Class	39	∅			∅		
Data Class	52	24	18/24 (75%)	18/52 (34,62%)	55	41/55 (74,55%)	41/52 (78,85%)
Speculative Generality	8	9	8/9 (88,88%)	8/8 (100%)	∅		
The Coupler							
Message Chains	8	∅			∅		

TAB. 5.2: Résultat de la détection automatique pour les code-smells

		IPlasma			PtiDej		
Nom de l'anti-patrons	**Réels défauts**	Défauts détectés	Précision	Rappel	Défauts détectés	Précision	Rappel
The Blob	8	11	8/11 (72,72%)	8/8 (100%)	10	7/10 (70%)	7/8 (87,5%)
Functional Decomposition	18	∅			16	8/16 (50%)	8/18 (44,44%)
Spaghetti Code	12	∅			13	9/13 (69,23%)	9/12 (75%)
Swiss Army Knife	8	∅			8	6/8 (75%)	6/8 (75%)

TAB. 5.3: Résultat de la détection automatique pour les anti-patrons

5.5 Interprétation des résultats

Au premier abord, en analysant les chiffres des tableau 5.2 et 5.3, on pourrait penser que iPlasma est plus performant que Ptidej, même si le terme "performant" n'est pas adéquat. Cette déduction n'est absolument pas recevable puisque la seule méthode pour comparer des outils de détection est d'utiliser leur valeur de précision et de rappel. Comparer les outils sur le nombre de type de défauts qu'ils détectent n'est pas non plus significatif. En effet il vaut mieux détecter peu de défauts et obtenir une précision et un rappel satisfaisant plutôt que l'inverse.

Néanmoins nous pouvons remarquer que iPlasma semble être plus efficace sur la détection de code-smells alors que Ptidej semble être plus efficace sur celle des anti-patrons. Toutefois, comme nous l'avons dit précédemment, Ptidej est capable de détecter la quasi totalité des code-smells mais il faut coder "en dur" les règles de détection. Nous rappelons qu'il ne nous était pas demandé de définir des règles pour Ptidej en utilisant son API.

5.5.1 Différences entre la détection manuelle et automatique

Il est intéressant de se demander si l'utilisation d'outils est une base sûre pour la détection de défauts. Il est vrai que, tous outils confondus, la détection manuelle reste la plus fiable. Néanmoins, il n'est pas concevable de passer autant de temps sur la détection des défauts d'une application. Pour GanttProject, nous avons passé une vingtaine d'heures pour trouver tous les défauts mais cela nous aurait pris dix fois plus de temps pour lister tous les défauts présents dans Eclipse, par exemple. De plus, en effectuant la détection manuelle, nous nous sommes bien rendu compte que le processus de détection était systématique et fastidieux et donc qu'un programme informatique était parfaitement adapté à ce type de traitement. Il est intéressant de noter que le temps d'apprentissage de la méthodologie manuelle et le temps de prise en main est relativement le même. En revanche, le temps que l'humain passe pour inspecter une classe équivaut au temps mis par la machine pour analyser un projet comme GanttProject.

Dans le cadre de notre expérimentation, nous avons supposé que la détection effectuée par l'homme était fiable à 100%. Les résultats fournis par les outils n'ont pas contredis cette hypothèse. La principale différence entre une détection faite par l'homme et une détection faite par la machine est que l'homme est doté d'un sens critique. Nous avons remarqué aussi que les outils n'ont jamais trouvé de défauts oubliés lors de la détection manuelle. Il serait néanmoins normal que cela se produise puisque la fiabilité de l'homme est relative. Par contre, les outils ne sont pas encore capables d'atteindre un niveau de granularité aussi élevé que l'homme. C'est pour cela que les outils de détection peuvent qualifier de défaut un objet qui n'en est pas un.

Il est vrai que la première chose que l'on remarque, lorsque l'on regarde les tableaux, est que Ptidej semble ne pas détecter beaucoup de défauts. Cela est principalement dû au fait qu'un certain nombre de

défauts n'a pas été implanté dans l'interface graphique et donc qu'il fallait écrire les règles de détection en Java pour que ces défauts soient intégrés à l'outil. Néanmoins, nous pouvons remarquer aussi que iPlasma propose de détecter certains défauts alors qu'il n'en trouve pas.

5.5.2 Différences entre iPlasma et Ptidej

La principale différence entre les deux outils est causée par l'absence de spécification. En effet, le premier travail que nous avions réalisé dans ce projet était de proposer une définition de chaque défauts de conception. Il y a peu de chance que les concepteurs de iPlasma et de Ptidej aient choisi les mêmes définitions que nous, et que iPlasma propose les mêmes abstractions que Ptidej et vice-versa. A priori, nous pouvons penser que iPlasma se rapproche de notre définition des code-smells, tandis que Ptidej se rapproche de notre définition des anti-patrons. Ptidej permet de détecter les quatre anti-patrons avec une précision comprise entre 50% et 75% et un rappel compris entre 44% et 75%. Au vu de ces résultats, il y a une forte probabilité pour que Ptidej vérifie notre heuristique concernant les anti-patrons, à savoir qu'ils sont basés sur l'existence de code-smells. De plus, les trois code-smells que nous avons pu détecter avec Ptidej ont obtenu un bon rappel (supérieur à 75%). Parmi ces trois défauts, selon notre définition, les défauts Data Class et Large Class constituent l'anti-patron Blob.

Il aurait été intéressant que Ptidej intègre plus de règles de détection de code-smells car iPlasma semble être un sérieux concurrent. En effet, il propose de détecter 9 code-smells sur les 17. Parmi ces neuf défauts, quatre détections (pour Long Method, Long Parameter List, Poor Usage of Abstract Class et Speculative Generality) ont obtenu de très bons résultats (entre 95% et 100% pour le rappel). Ceci s'explique par le fait que les règles sont facilement personnalisables et adaptables à nos définitions. Pour la détection du défaut Large Class et Lazy Class, cela vient du fait que iPlasma ne permet pas de définir des filtres basés sur la somme de métriques. En effet, pour Large Class par exemple, il aurait fallu chercher les classes ayant un nombre d'attributs et de méthodes supérieur à une certaine valeur seuil (NOA + NOM $\geq$ SEUIL) mais l'éditeur de filtres de iPlasma ne propose pas les opérateurs arithmétiques.

Le résultat de la détection du défaut Temporary Field est erroné. En effet, l'outil comptabilise des entités que nous avons défini comme Data Class. L'outil considère que les attributs des Data Classes sont des Temporary Fields car ils ne sont utilisés que dans les getters/setters ; hypothèse que nous n'avons pas émise.

5.6 Avantages et limitations des outils

Les deux outils sont composés d'une interface graphique intuitive. Malgré cette simplicité "graphique", nous avons remarqué que IPlasma restait difficile à prendre en main. Le problème avec iPlasma est que son utilisation s'articule autour de ses trois sous-menus ("Group", "Property" et "Filter") qui sont complètement désorganisés. En effet, les notions manipulables dans les différents menus ne sont pas regroupées de façon logique (hormis l'ordre lexicographique). On ne rencontre pas ce problème avec Ptidej car les éléments sont regroupés sous des onglets et les noms utilisés pour désigner les différentes notions sont suffisamment explicites pour que l'utilisateur "novice" s'y retrouve. Pour ce qui est de l'utilisation de l'espace graphique, les deux outils proposent une disposition claire des fonctionnalités. Comme mentionné plus haut, IPlasma affiche les résultats d'une détection sous la forme d'une liste tandis que Ptidej les fait apparaître sur le diagramme de classe. Il est néanmoins possible avec Ptidej d'obtenir la liste des classes sur lesquelles il y a un "focus" tandis que, à l'inverse, iPlasma ne permet pas d'obtenir de diagramme de classe.

En ce qui concerne l'exploitation du modèle, iPlasma est le seul à considérer, en plus des classes "traditionnelles", les classes internes et les classes anonymes. Par contre, IPlasma identifie les classes anonymes seulement par un numéro et ne fournit aucune information permettant de les localiser dans l'architecture logicielle.

Les deux outils sont confrontés à des problèmes de stabilité et de consommation abusive des ressources. Ils sont écrits tous les deux en Java et nous trouvons regrettable qu'ils soient diffusés sous forme d'exécutables Windows.

De manière générale, nous avons pu noter que le principal défaut des deux outils est qu'ils ne sont pas fournis avec leur documentation "utilisateur". Leur bonne utilisation est donc uniquement basée sur les pré-requis comme la notion de métriques.

6- Conclusion

Les solutions de conceptions telles que les patrons de conception sont apparues de l'émergence des langages objets. Ces mêmes solutions ont donné naissance à des défauts de conception tels que les code-smells et les anti-patrons de conception. Ce projet nous a été proposé afin d'étudier la pertinence des outils de détection automatique des défauts de conception dans des architectures orientées objet. Nous nous sommes intéressés aux outils Ptidej et iPlasma.

En l'absence de spécification et de formalisme des défauts de conception, nous avons tout d'abord dressé la liste de défauts qu'il nous semblait intéressant d'étudier et nous avons proposé une définition pour chacun d'eux.

Comme dans toutes expériences, pour pouvoir comparer deux choses, il nous fallait un témoin, une référence. Dans le cadre de la détection de défaut, nous sommes partis du principe que l'homme était infaillible et que lui seul l'homme était à même de détecter tous les défauts de conception d'un programme. Nous avons décomposé notre témoin en deux aspects. Tout d'abord, la définition d'une méthodologie de détection manuelle et ensuite les résultats de cette méthodologie. La méthodologie a été élaborée à partir de la définition que nous avons proposée en premier lieu. Nous l'avons ensuite appliquée sur le projet open-source GanttProject. Nous avons dénombré 452 défauts (code-smells et anti-patrons confondus) parmi les 240 classes du projet. Ces défauts ont été répertoriés dans deux fichiers XML afin de compléter notre base de connaissance de défauts de conception dans les architectures objet.

Une fois le relevé manuel effectué, nous avons pu commencer la comparaison des outils en calculant pour chacun d'entre eux leur précision et leur rappel. La précision permet de mesurer à quel degré un outil de détection fournit des résultats corrects, alors que le rappel permet de mesurer si l'outil est à même de détecter tous les défauts existants.

Nous avons enfin terminé par l'interprétation des résultats dans le but de faire une critique de la détection manuelle face à la détection automatique et enfin de l'outil iPlasma et Ptidej. Même si l'analyse manuelle reste la plus fiable, car l'homme est capable de prendre de la distance face à la rigidité de la méthodologie, il s'avère que les outils de détection permettent une détection efficace à partir du moment où ils sont pris correctement en main. L'outil iPlasma, plus adapté à la détection de code-smells, nous a paru intéressant avec son éditeur de filtres. En effet, les notions de regroupement et de filtrage se sont avérées pertinentes pour isoler les sous-ensembles d'entités défectueuses. Ptidej a prouvé son efficacité en obtenant une bonne précision et un bon rappel sur la détection des anti-patrons. En ce qui concerne la détection de code-smells, les règles que l'on peut spécifier grâce à l'API sont en cours d'intégration dans l'interface graphique.

Certains points du projet auraient mérité d'être approfondis comme, par exemple, la détermination des valeurs seuil qui peut être considérée comme un travail à part entière.

Bilan personnel

Voulant, depuis le début de mes études en informatique, poursuivre vers la filière recherche, j'attendais avec impatience ce projet. Lorsque Anne-Françoise Le Meur m'a parlé du projet que Naouel Moha proposait, j'ai tout d'abord émis quelques réserves. En effet, habitué à toujours produire du code lors des projets, la finalité de ce projet de Master n'était pas le développement d'un logiciel. Paradoxalement, c'est la principale raison pour laquelle j'ai pris le sujet. En me présentant le sujet, Naouel Moha m'a dit que je suivrais une démarche typiquement "recherche" : documentation, proposition, expérimentation et interprétation. Un fois le projet lancé, nous nous sommes fixés un calendrier de livrables. En effet, une partie de mon travail (spécifications des défauts) a servi des bases pour le projet d'autres étudiants. Cela fut très valorisant pour moi. Au mois d'avril, j'ai eu l'occasion de rencontrer le directeur de thèse de Naouel Moha, Yann-Gaël Guéhéneuc. Cette rencontre nous a permis, à Naouel Moha et à moi-même, de faire le point avec lui sur l'avancement du projet. Satisfait de mon travail, Yann-Gaël Guéhéneuc m'a proposé de poursuivre mon projet en stage au sein de son équipe à Montréal. Malheureusement ce stage n'a pu avoir lieu à cause d'un problème de quota de permis de travail.

Un autre aspect que j'ai apprécié lors de ce travail est le dialogue privilégié qui s'est installé avec mes divers encadrants. En effet, pour nous étudiants, cette collaboration étroite avac nos enseignants est très enrichissante. Il m'est même arrivé d'avoir l'impression de ne plus être qu'un simple étudiant mais un collègue ! Enfin, ce projet m'a appris à m'organiser dans mon travail, en rédigeant ce document de façon incrémentale et en devant respecter des deadlines. Au niveau de mes connaissances, j'ai acquis de bonnes notions dans le domaine défauts de conception et de leur détection. D'ailleurs, je le ressens déjà dans ma façon de programmer.

ANNEXES

A- DTD du fichier codesmells_gantt.xml

```
<!ELEMENT codesmells ( program )>

<!ELEMENT program ( name | location | codesmell )*>
<!ATTLIST program type CDATA #REQUIRED>

<!ELEMENT name ( #PCDATA )>

<!ELEMENT location ( #PCDATA )>

<!ELEMENT codesmell ( microArchitectures )*>
<!ATTLIST codesmell name CDATA #REQUIRED>

<!ELEMENT microArchitectures ( microArchitecture )*>

<!ELEMENT microArchitecture ( roles )>
<!ATTLIST microArchitecture number CDATA #REQUIRED>

<!ELEMENT roles ( longMethods | largeClasses | lazyClasses
                | dataClasses | tooMuchInformationsHiding | notEnoughInformationsHiding
                | dataClumps | switchStatements | refusedBequests
                | messageChains | alternativeClasses | longParameterLists
                | temporaryFields | poorUsagesOfInterfaces | primitiveObsessions
                | poorUsagesOfAbstractClasses | speculativeGeneralities)*>

<!ELEMENT longMethods ( longMethod )>
<!ELEMENT longMethod ( entity | comment )*>

<!ELEMENT largeClasses ( largeClass )>
<!ELEMENT largeClass ( entity | comment )*>

<!ELEMENT lazyClasses ( lazyClass )>
<!ELEMENT lazyClass ( entity | comment )*>

<!ELEMENT dataClasses ( dataClass )>
<!ELEMENT dataClass ( entity | comment )*>

<!ELEMENT tooMuchInformationsHiding ( tooMuchInformationHiding )>
<!ELEMENT tooMuchInformationHiding ( entity | comment )*>

<!ELEMENT notEnoughInformationsHiding ( notEnoughInformationHiding )>
<!ELEMENT notEnoughInformationHiding ( entity | comment )*>

<!ELEMENT dataClumps ( dataClump )>
<!ELEMENT dataClump ( entity | comment )*>

<!ELEMENT switchStatements ( switchStatement )>
<!ELEMENT switchStatement ( entity | comment )*>

<!ELEMENT refusedBequests ( refusedBequest )>
<!ELEMENT refusedBequest ( entity | comment )*>

<!ELEMENT messageChains ( messageChain )>
<!ELEMENT messageChain ( entity | comment )*>

<!ELEMENT alternativeClasses ( alternativeClass )>
<!ELEMENT alternativeClass ( entity | comment )*>

<!ELEMENT longParameterLists ( longParameterList )>
<!ELEMENT longParameterList ( entity | comment )*>

<!ELEMENT temporaryFields ( temporaryField )>
<!ELEMENT temporaryField ( entity | comment )*>

```

```
64 <!ELEMENT poorUsagesOfInterfaces (poorUsageOfInterfaces )>
65 <!ELEMENT poorUsageOfInterfaces ( entity | comment )*>
66
67 <!ELEMENT poorUsagesOfAbstractClasses (poorUsageOfAbstractClasses )>
68 <!ELEMENT poorUsageOfAbstractClasses ( entity | comment )*>
69
70 <!ELEMENT speculativeGeneralities (speculativeGenerality )>
71 <!ELEMENT speculativeGenerality ( entity | comment )*>
72
73 <!ELEMENT primitiveObsessions (primitiveObsession )>
74 <!ELEMENT primitiveObsession ( entity | comment )*>
75
76 <!ELEMENT entity ( #PCDATA )>
77 <!ELEMENT comment ( #PCDATA )>
```

B- DTD du fichier antipatterns_gantt.xml

```
<!ELEMENT antipatterns ( program )>

<!ELEMENT program ( name | location | antipattern )*>
<!ATTLIST program type CDATA #REQUIRED>

<!ELEMENT name ( #PCDATA )>
<!ELEMENT location ( #PCDATA )>

<!ELEMENT antipattern ( microArchitectures )*>
<!ATTLIST antipattern name CDATA #REQUIRED>

<!ELEMENT microArchitectures ( microArchitecture )*>

<!ELEMENT microArchitecture ( roles )*>
<!ATTLIST microArchitecture number CDATA #REQUIRED>

<!ELEMENT roles ( swissArmyKnives | functionalDecompositions | blobs | spaghettiCodes )*>

<!ELEMENT swissArmyKnives ( swissArmyKnife )>
<!ELEMENT swissArmyKnife ( entity | comment)*>

<!ELEMENT functionalDecompositions ( functionalDecomposition )>
<!ELEMENT functionalDecomposition ( entity | comment )*>

<!ELEMENT blobs (blob) >
<!ELEMENT blob ( entity | comment)*>

<!ELEMENT spaghettiCodes (spaghettiCode)>
<!ELEMENT spaghettiCode (entity | comment)*>

<!ELEMENT entity ( #PCDATA )>
<!ELEMENT comment ( #PCDATA )>
```

C- La grammaire BNF

```
/*------------------------------ RULES ------------------------------*/

set_rules        ::=  rule_card | set_rules rule_card

rule_card        ::=  RULE_CARD: string { list_rules } ;

list_rules       ::=  rule | list_rules rule

rule             ::=  RULE: string { content_rule } ; | RULE: string ;

content_rule     ::=  operator string string | list_relationships | list_attributes

operator         ::=  INTER | UNION | DIFF | INCL | NEG

/*---------------------------- ATTRIBUTES ----------------------------*/

list_attributes  ::=  attribute | operator attribute attribute

attribute        ::=  ( METRIC: id_metric , value_ordi )
                  |   ( SEMANTIC: id_semantic , value_semantic )
                  |   ( STRUCT: id_struct , value_struct )

id_metric        ::=  id_metric + id_metric | id_metric - id_metric | string

value_ordi       ::=  VERY_HIGH | HIGH | MEDIUM | LOW | VERY_LOW | NONE

id_semantic      ::=  CLASSNAME | METHODNAME | FIELDNAME

value_semantic   ::=  { cont_value_semantic }

cont_value_semantic      ::=  string | string , cont_value_semantic

id_struct        ::=  CLASS | INTERFACE | METHOD | FIELD | PARAMETER | COMMENTS

value_struct     ::=  string

/*-------------------------- RELATIONSHIPS ---------------------------*/

list_relationships ::=  relationship | relationship list_relationships

relationship       ::=  name_relation: string FROM: string cardinality TO: string cardinality

name_relation      ::=  ASSOC | AGGREG | COMPOS

cardinality        ::=  ONE | MANY | ONE_OR_MANY | OPTIONALLY_ONE
```

D- La rule card pour la détection du défaut Long Method

```
1 RULE_CARD : LongMethod
2 {
3         RULE : LongMethod { (METRIC: METHOD_LOC, 35, 0) };
4 };
```

E- La rule card pour la détection du défaut Large Class

```
1 RULE_CARD : LargeClass
2 {
3         RULE : LargeClass { (METRIC: NMD + NAD, 43, 0) };
4 };
```

Bibliographie

[1] William J. Brown, Raphael C. Malveau, Hays W. McCormick III, Thomas, and Inc. J. Mowbray, John Wiley & Sons. *AntiPatterns – Refactoring Software, Architectures, and Projects in Crisis*. Robert Ipsen, 1998.

[2] Martin Fowler. *Refactoring – Improving the Design of Existing Code*. Addison-Wesley, 1999.

[3] W. B. Frakes and R. Baeza-Yates. *Information Retrieval : Data Structures and Algorithms*. Prentice-Hall, 1992.

[4] Erich Gamma, Richard Helm, and Ralph Johnson andJohn Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[5] Lanza Marinescu. *Object-oriented Metrics in Practice*. Springer, 2005.

[6] Naouel Moha. *Detection and Correction of Design Defects in Object-Oriented Architectures*. Research proposal submitted in partial fulfilment of the requirements for the degree of doctor of philosophy (Ph.D.) in computer science, December 2006.

[7] Mika Mäntylä, Jari Vanhanen, and Casper Lassenius. A taxonomy and an initial empirical study of bad smells in code. In *ICSM '03 : Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society.