Une taxonomie et un métamodèle pour la détection des défauts de conception

Naouel Moha — Duc-loc Huynh — Yann-Gaël Guéhéneuc

Équipe PTIDEJ
GEODES – Groupe de recherche sur les systèmes ouverts et distribués et en génie logiciel expérimental
Université de Montréal
CP 6128 succ. Centre Ville
Montréal, Québec, H3C 3J7, Canada
{mohanaou, huynhduc, guehene}@iro.umontreal.ca

RÉSUMÉ. *Les défauts de conception sont à rapprocher des patrons de conception qui sont aujourd'hui largement utilisés : les patrons de conception proposent de "bonnes" solutions à des problèmes récurrents dans les architectures à objets, tandis que les défauts de conception sont de "mauvaises" solutions. Cependant, contrairement aux patrons, les défauts de conception n'ont pas de représentation précise et structurée et existent seulement sous la forme de descriptions textuelles sujettes à interprétation, qui ne permettent pas leur détection et leur correction précise et efficace. Nous proposons une méthodologie pour représenter les défauts de conception basée sur un métamodèle à partir d'une taxonomie des défauts. Nous appliquons et validons cette méthodologie sur un ensemble de défauts de conception, tels le Blob et le Swiss Army Knife.*

ABSTRACT. *Design defects are similar to design patterns, which are today largely used: design patterns propose "good" solutions to recurring design problems in object-oriented architectures, whereas design defects are "bad" solutions. However, unlike design patterns, design defects have not yet been widely studied and are based essentially on textual descriptions prone to interpretation. The lack of precise and structured representation of design defects hinders the efficient detection and correction of these defects. We propose a methodology to represent design defects based on a meta-model using a taxonomy of defects. We apply and validate this methodology on a set of design defects such as the Blob and the Swiss Army Knife.*

MOTS-CLÉS : *Défauts de conception, anti-patrons, métamodèle, architectures orientées-objet.*

KEYWORDS: *Design defects, antipatterns, metamodel, object-oriented architectures.*

1. Introduction

La détection et la correction des défauts de conception sont des enjeux importants pour améliorer la qualité des architectures à objets et pour en faciliter l'évolution et la maintenance. Une architecture dépourvue de défauts réduit de manière significative les coûts de maintenance en facilitant la compréhension et les changements. Une bonne architecture est donc particulièrement importante car la maintenance est l'une des phases les plus coûteuses du cycle de vie des logiciels.

Les patrons de conception [GAM 94] aident à construire des programmes flexibles, réutilisables et "élégants" en répondant à des problèmes récurrents de conception. Pourtant, les programmes restent encore trop souvent difficiles à maintenir à cause de mauvaises pratiques de conception. Les *défauts de conception* sont de mauvaises pratiques de conception des programmes, qui englobent les mauvaises solutions à des problèmes récurrents dans les architectures à objets (par opposition aux patrons de conception [GUÉ 01]), telles que les anti-patrons, et les défauts liés à l'utilisation abusive des patrons de conception.

Par *défaut*, nous faisons référence à une faute dans le modèle causal classique faute–erreur–défaillance. Une faute peut entraîner une erreur et une défaillance est une propagation potentielle de cette erreur à l'extérieur du programme. Ainsi, nous considérons les défauts comme des fautes qui apparaissent au cours de la vie du logiciel et qui, par leur présence dans une architecture, peuvent causer des défaillances du programme ou des problèmes de maintenance. Les défauts sont concrètement constitués d'un ensemble de symptômes, ou *mauvaises odeurs*, qui représentent des problèmes concrets dans une architecture.

La terminologie liée aux défauts de conception n'est pas clairement définie : les termes défauts, anti-patrons, problèmes, mauvaises odeurs sont parfois utilisés en complément ou en remplacement les uns des autres alors qu'ils ne décrivent pas la même granularité de défauts. De plus, notre objectif à long terme est de développer des outils qui permettent de détecter et de corriger semi-automatiquement des défauts dans des architectures à objets (voir [MOH 05b]).

C'est pourquoi, nous proposons une représentation précise et structurée des défauts de conception. Cette représentation est basée sur un métamodèle, comme les patrons de conception (par exemple [ALB 02]). Un métamodèle des défauts de conception est une solution au problème de leur interprétation car il limite les ambiguïtés, structure les concepts et encourage la construction d'algorithmes et d'outils. Nous ne prétendons pas que *notre* représentation des défauts est définitive mais que notre métamodèle, par la réification des concepts liés aux défauts, limite les problèmes d'interprétation et facilite le développement d'algorithmes et d'outils. À notre connaissance, aucun métamodèle n'existe pour représenter les défauts. Avant de concevoir notre métamodèle, nous définissons une taxonomie des défauts. Cette taxonomie qui regroupe d'une part la terminologie des défauts incluant les anti-patrons, les défauts de patron et les mauvaises odeurs et d'autre part leur classification a pour but de définir, comparer et classer les défauts afin d'en fournir une description précise et structurée.

Les contributions de cet article sont décrites une à une dans les différentes sections de celui-ci : dans la section 2, nous présentons une méthodologie pour obtenir une représentation précise et structurée des défauts de conception. Dans la section 3, nous présentons une taxonomie et une classification des différents défauts de conception de la littérature. Dans la section 4, nous introduisons un métamodèle pour représenter les défauts de conception. Enfin, dans la section 5, nous appliquons et validons notre méthodologie sur un ensemble de défauts et de concepts associés : Swiss Army Knife, Shotgun Surgery, Divergent Change et Observer. La section 6 conclut cet article et présente les travaux futurs.

2. Méthodologie

Nous proposons comme première contribution une méthodologie en 5 phases pour concevoir une représentation précise et structurée des défauts de conception et instancier ce métamodèle pour différents défauts (*cf.* figure 1). D'après notre connaissance, aucune méthodologie semblable n'a été proposée pour la représentation des défauts de conception. Nous illustrons notre méthodologie avec le défaut Blob. Le Blob [BRO 98] (aussi appelé God class [RIE 96]) correspond à une large classe complexe qui monopolise tout le traitement réalisé dans un programme et qui est entourée de simples classes de données. Un exemple type de Blob sont les classes 'main' ou 'manager' dans les programmes orientés objets.

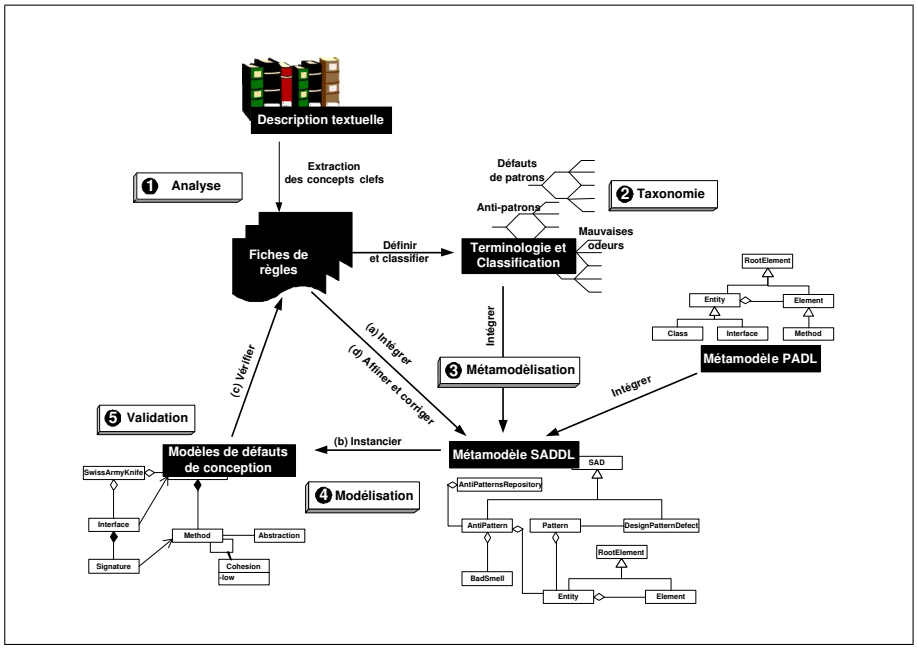


Figure 1. *Méthodologie pour la représentation des défauts de conception*

2.1. *Phase 1 : analyse*

La phase d'analyse est manuelle et consiste à extraire les concepts clefs des descriptions textuelles décrivant les défauts de conception. Les concepts clefs forment un vocabulaire de concepts réutilisables avec lesquels nous décrivons les défauts de conception. Les descriptions des défauts de conception proviennent essentiellement de la littérature [RIE 96 ; BRO 98 ; DEM 02 ; DUD 03]. Les concepts clefs de chaque défaut sont identifiés et organisés sous forme d'ensembles de règles. Les règles identifient les rôles prépondérants dans les défauts, leurs relations et leurs attributs internes (tels que la taille, la complexité, la cohésion, etc.). Les attributs sont essentiellement spécifiés sous forme de métriques. L'ensemble de ces règles et leurs relations constituent *une fiche de règles*.

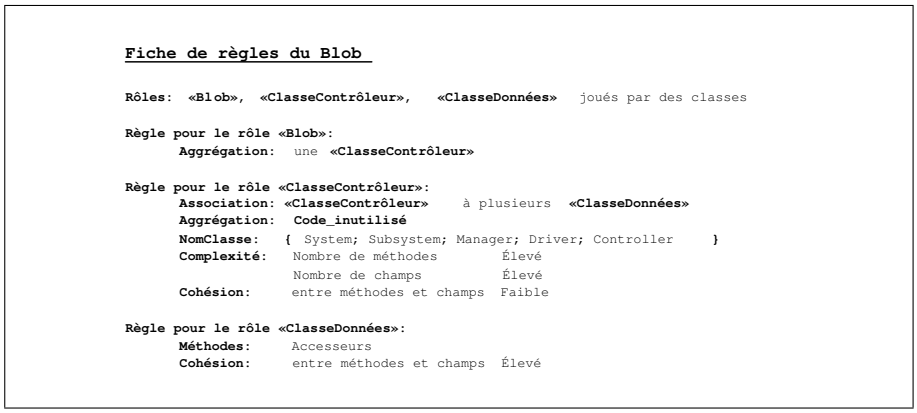
La spécification des défauts sous forme de règles nécessite une recherche exhaustive de leurs définitions dans la littérature, une compilation de ces définitions pour supprimer les synonymes, homonymes et lever les ambiguïtés et, enfin, une étude de chaque défaut en rapport aux autres, pour obtenir un ensemble minimal de concepts clefs avec lequel construire les règles. Les règles permettent ainsi de passer d'une description littéraire à une description synthétique qui élimine les incohérences et les redondances et facilite l'identification des relations entre défauts de conception. Cette phase est à rapprocher de la phase de construction des ontologies. La structure des fiches de règles a été inspirée des règles définies dans les algorithmes d'apprentissage par règles propositionnelles.

La figure 2 donne la fiche de règles du Blob. Le Blob est composé d'une large classe associée à plusieurs classes de données. L'analyse des concepts clefs nous a permis de déduire que la large classe joue le rôle de classe contrôleur (ou système, sous-système, manager, pilote). Une classe de données est composée de champs et de méthodes d'accès fortement cohésives. La large classe peut contenir du code mort et est composée de nombreux champs et méthodes avec une faible cohésion.

Cette phase d'analyse est itérative car à l'ajout d'une nouvelle description textuelle d'un défaut de conception, nous extrayons les concepts clefs de celui-ci et les comparons avec les concepts déjà identifiés pour assurer leur cohérence. De par notre expérience (*cf.* la section 5 sur la modélisation et validation), le nombre de concepts clefs nécessaire à la description des défauts de la littérature est fini et de taille raisonnable. Nous avons trouvé une vingtaine de concepts clefs pour décrire une quinzaine de défauts de conception.

2.2. *Phase 2 : taxonomie*

La définition d'une taxonomie des défauts de conception est la seconde étape nécessaire à leur représentation précise et structurée (*cf.* figure 1). Une taxonomie est une organisation des concepts clefs sous forme de catégories disjointes et leur organisation sous forme d'un réseau faisant apparaître clairement leurs points communs et leurs différences. La taxonomie des défauts inclut leur terminologie et une classifica-



Fiche de règles du Blob

Rôles: **«Blob»**, **«ClasseContrôleur»**, **«ClasseDonnées»** joués par des classes

Règle pour le rôle «Blob»:
Aggrégation: une **«ClasseContrôleur»**

Règle pour le rôle «ClasseContrôleur»:
Association: **«ClasseContrôleur»** à plusieurs **«ClasseDonnées»**
Aggrégation: **Code_inutilisé**
NomClasse: **{** System; Subsystem; Manager; Driver; Controller **}**
Complexité: Nombre de méthodes Élevé
Nombre de champs Élevé
Cohésion: entre méthodes et champs Faible

Règle pour le rôle «ClasseDonnées»:
Méthodes: Accesseurs
Cohésion: entre méthodes et champs Élevé

Figure 2. *Fiche de règles du Blob*

tion par leur concepts clefs afin d'obtenir un modèle de référence pour les distinguer les uns des autres et faire ressortir leurs particularités. Elle définit les termes associés aux défauts pour faciliter leur identification et limiter les problèmes d'interprétation. Par exemple, Trifu et Marinescu [TRI 03] proposent une détection de certains défauts de conception qui sont réfutés par Munro [MUN 05] sur une différence d'interprétation des défauts. La classification s'appuie sur les règles définies à la phase 1 d'analyse et permet de mettre en perspective les défauts les uns par rapport aux autres. Par exemple, le Blob est un anti-patron, les anti-patrons sont une catégorie de défauts.

2.3. *Phase 3 : métamodélisation*

La troisième phase consiste à concevoir un métamodèle pour décrire les défauts de conception sous une forme qui soit manipulable par l'ordinateur. Le métamodèle permet également d'obtenir des descriptions des défauts précises et structurées, similaires à celles des patrons de conception [ALB 02]. Un métamodèle est un langage qui fournit le vocabulaire nécessaire non seulement pour décrire les défauts mais aussi pour développer des techniques et outils de détection et de correction semi-automatiques. Notre métamodèle, SADDL (*Software Architectural Defects Description Language*), s'appuie sur les règles des défauts et sur leur taxonomie et étend le métamodèle PADL (*Pattern and Abstract-level Description Language*) [ALB 02], qui permet de décrire les programmes orientés-objet et les solutions des patrons de conception.

2.4. *Phase 4 : modélisation*

La phase 4 consiste à instancier le métamodèle SADDL pour décrire concrètement des défauts de conception de la littérature [RIE 96 ; BRO 98 ; DEM 02 ; DUD 03]. Nous concevons ainsi un catalogue de modèles de défauts qui servira de base aux techniques de détection et de correction semi-automatiques que nous développerons. La figure 3 décrit le modèle obtenu pour le Blob, qui correspond précisément à sa

fiche de règles (*cf.* figure 2). Nous avons utilisé une notation proche d'UML pour représenter graphiquement le modèle d'un défaut de conception issu de SADDL.

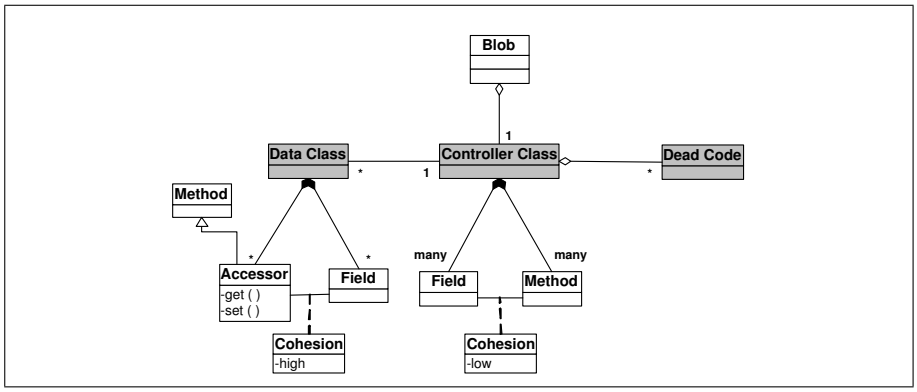


Figure 3. *Le modèle du Blob*

2.5. *Phase 5 : validation*

La phase 5 de validation consiste à s'assurer que les défauts de conception ont été décrits avec SADDL en accord avec les règles extraites dans la phase 1 (*cf.* figure 1). Elle consiste également à corriger et à enrichir SADDL pour préciser cette description. Cette phase est donc importante dans la méthodologie car elle permet d'améliorer l'analyse des concepts clefs, leur taxonomie, le métamodèle et la modélisation.

3. Phases 1 et 2 : analyse et taxonomie

Nous avons présenté une méthodologie de description des défauts de conception sous une forme précise et structurée, manipulable par l'ordinateur. Nous présentons maintenant en détails la phase 2 qui consiste à classifier les concepts clefs associés. Nous ne nous attardons pas sur la phase d'analyse, car les résultats de cette phase sont longs à décrire et correspondent globalement à l'extraction des concepts clefs et à la spécification des fiches de règles, mais présentons immédiatement une taxonomie incluant une terminologie et une classification des défauts. Cette taxonomie est une carte de l'état de l'art des défauts de conception, qui est la première tentative, à notre connaissance, de clarifier et de classifier les concepts clefs liés aux défauts.

Dans les défauts de conception, nous distinguons deux catégories principales : les défauts de patrons [MOH 05c], qui décrivent de mauvaises utilisations des patrons de conception, et les anti-patrons [BRO 98], qui décrivent de mauvaises solutions à des problèmes de conception. Les symptômes des défauts de conception (aussi appelés *mauvaises odeurs* ou *bad smells* en anglais) ne sont pas des défauts de conception, cependant, nous les incluons dans notre taxonomie car ils sont importants pour définir et distinguer certains défauts.

3.1. *Les défauts de patrons*

Terminologie. Les patrons de conception proposent de *bonnes* solutions à des problèmes de conception récurrents dans les architectures orientées-objets. Ces solutions sont mises en œuvre dans une architecture sous la forme de micro-architectures dont les constituants ont des structures et des organisations similaires à celles proposées par la solution du patron. Les défauts de patrons sont de mauvaises applications des solutions des patrons de conception [MOH 05c]. Il s'agit de déformation ou de dégradation des patrons de conception. Nous distinguons ainsi deux types de défauts liés aux patrons de conception, ou défauts de patrons : (1) les patrons déformés et (2) les patrons dégradés. Les patrons déformés sont des micro-architectures similaires mais non-identiques à celles proposées par les patrons de conception [GUÉ 01], qui, suite à des choix de conception, ont été remaniées dans leurs structures et organisations pour répondre à des contraintes imposées par l'architecture. Les patrons dégradés sont des micro-architectures similaires à des patrons de conception lors de mauvais choix de conception. Wendorff [WEN 01] définit ce type de défauts comme étant lié à des patrons de conception mal implantés.

Classification. La classification des défauts de patrons se base sur la classification de Gamma *et al.* (*cf.* figure 4). La catégorie `DesignPatternDefect` représente les défauts de patrons. Cette catégorie est liée aux patrons de conception dont elle est la représentation dégradée ou déformée. Cette catégorie est donc spécialisée en deux catégories `Degraded` et `Distorted`.

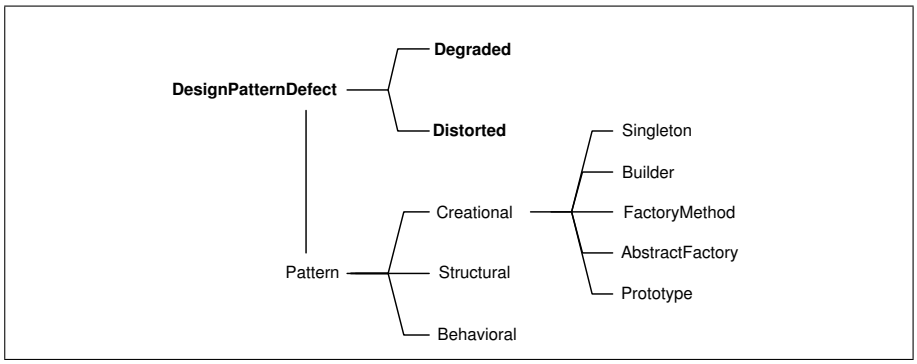


Figure 4. *Classification des défauts de patrons*

3.2. *Les anti-patrons*

Terminologie. Un anti-patron [BRO 98] est une forme littéraire décrivant une mauvaise solution à un problème de conception récurrent, dont l'utilisation a des effets négatifs sur la qualité. À l'inverse des patrons de conception, les anti-patrons décrivent ce qu'il *ne* faut *pas* faire. Il existe des anti-patrons généraux, tels que le Blob et le Swiss Army Knife, que nous décrivons dans la section 5.2, et des anti-patrons spécifiques à certains sous-domaines ou techniques du génie logiciel : J2EE [TAT 02 ;

DUD 03], XML [TAT 02], les processus concurrents [BOR 05], la performance [SMI 02].

Classification. Brown *et al.* [BRO 98] classifient les anti-patrons en trois catégories : les anti-patrons de développement, d'architecture et de gestion de projet ; nous nous intéressons uniquement aux deux premiers car ils représentent de mauvaises pratiques de conception. De plus, leur détection et leur correction améliorent la qualité des architectures orientés-objets et sont potentiellement possibles semi-automatiquement.

Parmi les anti-patrons, nous distinguons deux catégories : *intra-classe* si l'anti-patron se manifeste dans une classe, parmi les méthodes de la classe, et *inter-classe* dans l'architecture d'un programme, parmi plusieurs classes. Cette décomposition permet d'évaluer l'ampleur des changements suite à la correction des anti-patrons intra- et inter-classes. Nous décomposons encore ces deux catégories en : *structurel*, *sémantique*[1] et *comportemental*, si l'anti-patron est lié, respectivement, à la structure, à la sémantique ou au comportement d'un programme. Cette décomposition supplémentaire permet d'identifier les techniques à privilégier pour la détection des anti-patrons.

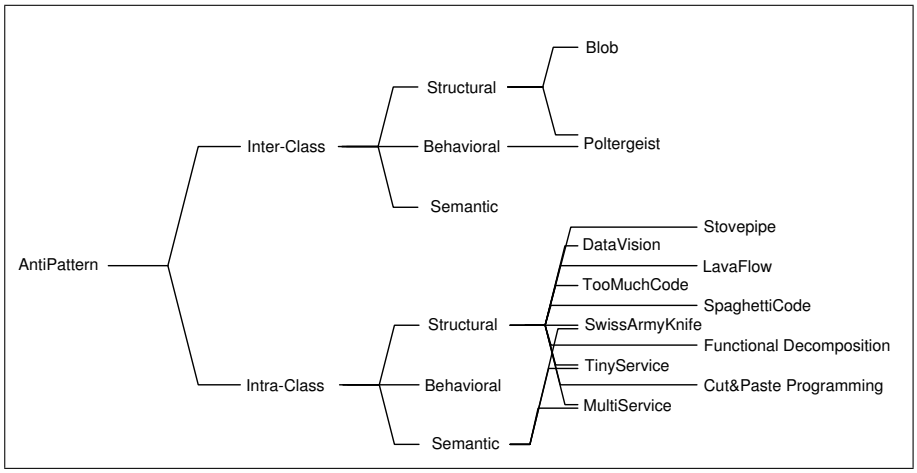


Figure 5. *Classification des anti-patrons*

3.3. *Les symptômes des défauts de conception*

Terminologie. Les symptômes des défauts de conception, ou *mauvaises odeurs* du code, ne sont pas des défauts car ils sont seulement des indicateurs de la présence possible de défauts. Beck et Fowler [FOW 99] décrivent ces symptômes comme des structures dans le code source qui suggèrent la possibilité d'une restructuration du code. Une restructuration est un changement effectué sur le code source d'un programme pour qu'il soit plus facile à comprendre et moins coûteux à maintenir, sans changer

1. La sémantique d'un programme est, par exemple, donnée par le sens des noms et des verbes utilisés pour nommer les constituants de son architecture.

son comportement externe. Du code dupliqué, de longues méthodes, de larges classes, de longues listes de paramètres, sont autant de symptômes de défauts de conception et d'opportunités de restructurations.

Classification. Fowler et Beck [FOW 99] présentent 22 symptômes de défauts de manière successive sans définir de catégories ou de relations entre eux. Aussi, il est difficile de pouvoir identifier ces symptômes, de les distinguer et, par conséquent, de les détecter dans le code source. Mäntylä [MAN 03] propose sept catégories de symptômes, telles que, par exemple, les concepts de l'orienté-objet ou les tailles importantes, comme de longues méthodes, de larges classes ou de longues listes de paramètres. Wake [WAK 03] distingue les symptômes apparaissant au sein d'une classe de ceux apparaissant entre plusieurs classes. Au sein d'une classe, par exemple, Wake distingue les symptômes que l'on peut mesurer de ceux liés aux données. Ces deux classifications se basent essentiellement sur la nature des symptômes alors que nous nous intéressons plus à leurs propriétés (structurelle, sémantique, comportementale) et leur étendue (intra- et inter-classes) dans le code source.

Comme Wake, nous distinguons les symptômes qui apparaissent au niveau d'une classe et ceux qui apparaissent entre plusieurs classes. Pour être cohérent avec la classification des anti-patrons et parce que nous considérons que la distinction entre défauts inter- et intra-classes reflète bien l'étendue des symptômes dans le code source, nous utilisons la même structure de classification que les anti-patrons (*cf.* figure 6). Notre classification est générique et permet de classer d'autres symptômes (*cf.* les catégories non grisées) hormis celles définies par Fowler et Beck (*cf.* les catégories grisées) et classer un symptôme dans plusieurs catégories (*e.g.* la catégorie `Duplicated code`). Il s'agit donc d'une amélioration des deux classifications existantes.

3.4. *Carte des défauts de conception*

Nous avons présenté précédemment la classification des défauts de conception. Cette classification a été décomposée en trois classifications disjointes sous forme d'arbres hiérarchiques afin d'offrir une vue synthétique des défauts de conception. La figure 7 présente une carte de ces défauts de conception avec tous les concepts clefs associés. À l'instar de la carte des patrons de conception proposée par Gamma *et al.* [GAM 94], sur cette carte[2], nous explicitons les relations entre les différents éléments de la classification des défauts de conception, en nous basant sur leurs similitudes. Pour éviter de surcharger la carte, nous ne présentons qu'un sous-ensemble d'anti-patrons. Cette carte décrit les liens entre les différents défauts (*cf.* les formes hexagonales), leurs relations avec les mauvaises odeurs (*cf.* les formes ovales en trait plein) et leurs propriétés (*cf.* les formes ovales en pointillés).

---

2. Bien que développée séparément, cette carte est à rapprocher des *correlation webs* [LAN 06] mais cette carte organise tous les concepts clefs identifiés, quels que soient leur granularité.

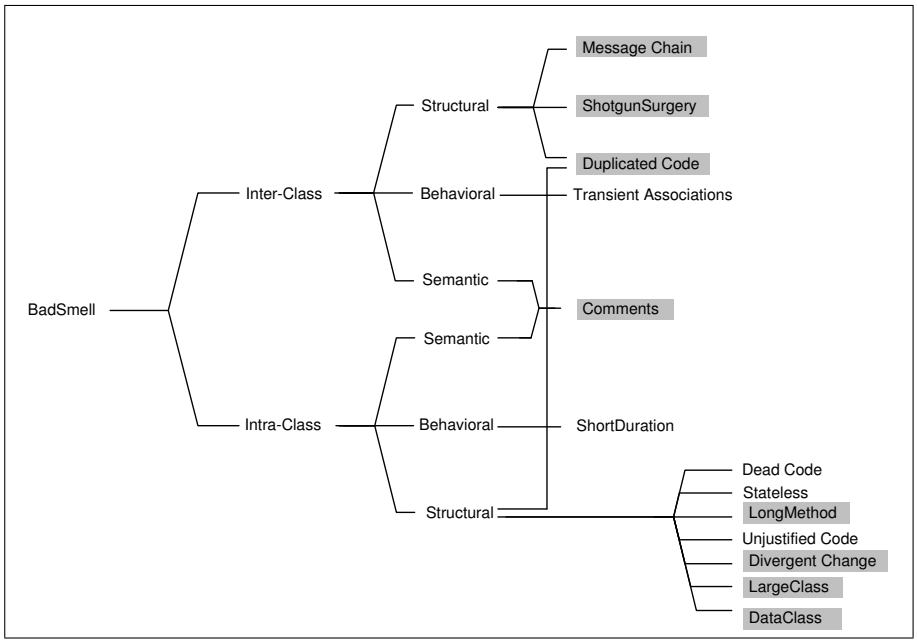


Figure 6. *Classification des mauvaises odeurs*

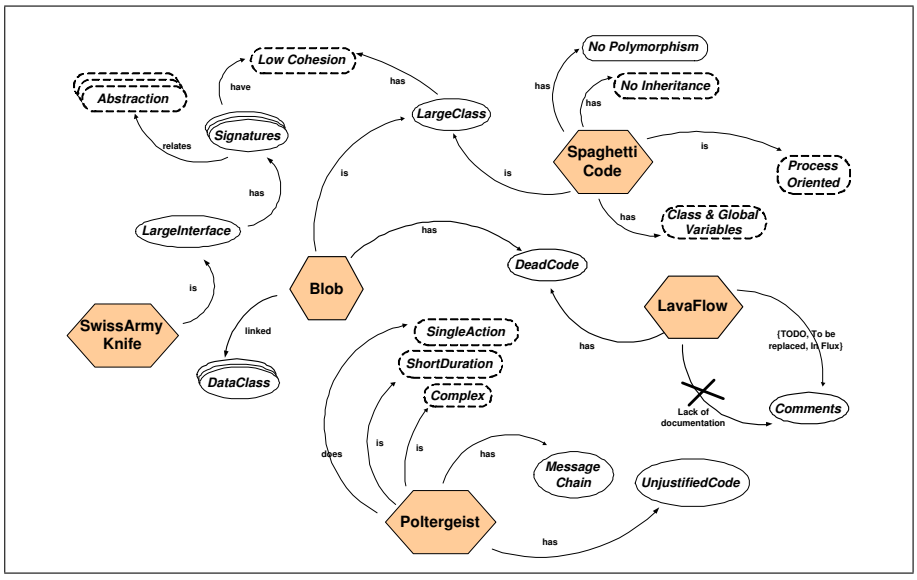


Figure 7. *Carte des défauts de conception*

4. Phase 3 : métamodélisation

De nombreux métamodèles ont été définis pour décrire les programmes orientés objets et les patrons de conception [TIC 00 ; ALB 02] mais, à notre connaissance, il

n'existe encore aucun métamodèle pour spécifier les défauts de conception. Le métamodèle FAMIX [TIC 00] fournit une représentation indépendante du langage de code source orienté-objet et supporte les techniques de restructuration mais ne permet de décrire précisément ni patrons ni défauts. Le métamodèle PADL [ALB 02] permet de décrire les programmes orientés-objets et certains patrons de conception mais ne possède pas tous les concepts clefs nécessaires pour la spécification des défauts.

4.1. *Le métamodèle PADL*

Le métamodèle PADL (*Pattern and Abstract-level Description Language*) permet de décrire des programmes orientés-objet et des solutions de patrons de conception [ALB 02]. Il propose un ensemble de constituants (entités : classes, interfaces ; et éléments : méthodes, champs, etc.) et définit les règles régissant leurs instanciations et compositions. Ces constituants permettent de décrire la structure et une partie du comportement d'un programme à partir de son code source et des solutions de patrons de conception.

À l'instar des patrons de conception, il est possible de décrire les défauts de conception car ceux-ci ont des concepts clefs communs : un défaut de conception est constitué d'entités et d'éléments, tels que des champs et des méthodes ; ces entités ont des liens d'association les unes avec les autres. D'autres concepts clefs pour la description des défauts de conception ne sont pas définis dans PADL, comme le couplage, la cohésion, les commentaires ou les méthodes d'accès. Aussi, nous étendons PADL pour décrire précisément les défauts de conception.

4.2. *Le métamodèle SADDL*

Le métamodèle SADDL étend PADL pour décrire les défauts de conception (*cf.* les classes grisées sur la figure 8). Il définit un certain nombre de constituants indépendants des langages propres aux défauts : la classe `SAD` représente les défauts et est spécialisée en deux sous-classes, `DesignPatternDefect` pour les défauts de patron et `AntiPattern` pour les anti-patrons. Cette hiérarchie est en accord avec la taxonomie des défauts de conception définie dans la section 3.

Les défauts de patrons sont associés à des patrons de conception (*cf.* la classe `Pattern`) dont ils sont des déformations ou dégradations. Un patron et un anti-patron sont des agrégations d'entités (*cf.* la classe `Entity`). Un anti-patron est un ensemble d'entités interagissant et ayant des caractéristiques particulières.

La classe `AntiPattern` décrit les anti-patrons. Un anti-patron est lié à des symptômes (*cf.* la classe `BadSmell`) car la présence de symptômes constitue une indication de la présence d'anti-patrons. Nous avons répertorié l'ensemble des anti-patrons dans un catalogue (*cf.* la classe `AntiPatternsRepository`). Par manque d'espace, nous ne présentons pas ici le catalogue complet des anti-patrons décrits avec SADDL, un rapport de recherche décrit plus d'une dizaine d'anti-patrons [MOH 05a].

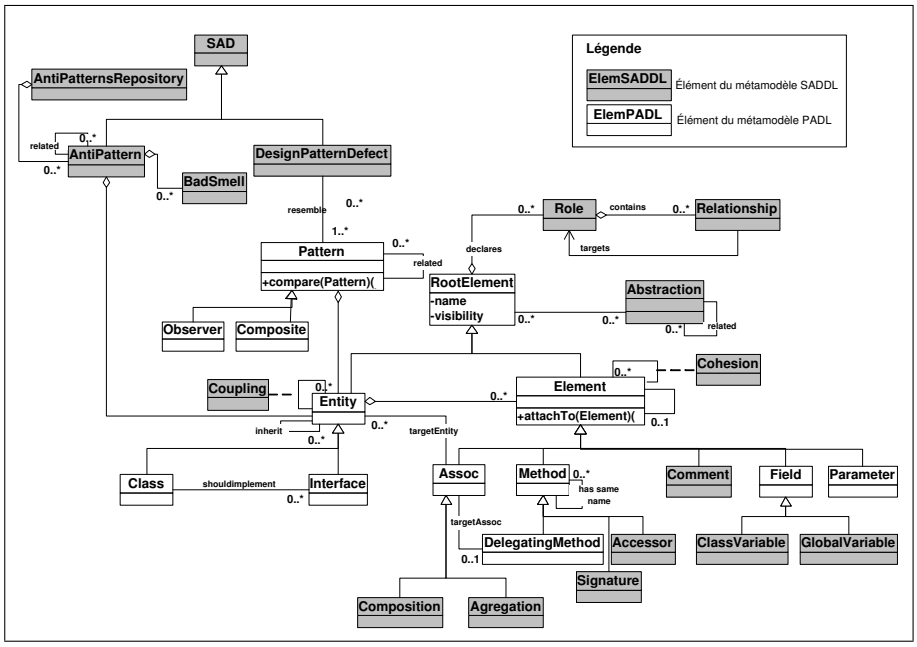


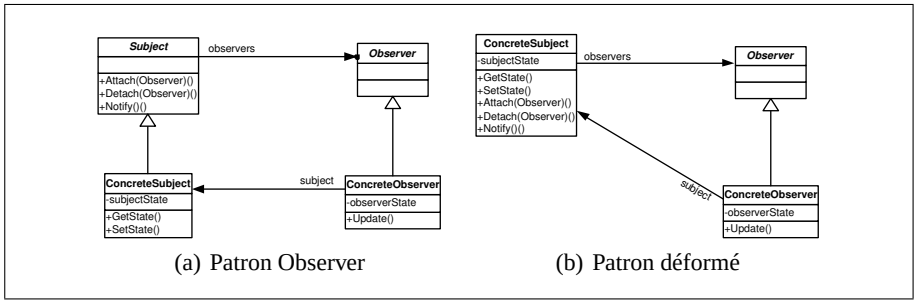
Figure 8. *Le métamodèle SADDL*

5. Phases 4 et 5 : modélisation et validation

La phase de modélisation consiste à instancier des défauts de conception à partir de SADDL sous forme de modèles. La phase de validation, quant à elle, consiste à vérifier que les modèles définis lors de la modélisation sont en accord avec les règles spécifiées lors de la phase 1 (*cf.* figure 1, étapes a, b, c, et d). Dans le cas où certains concepts clefs n'ont pas été spécifiés, ceux-ci sont intégrés à SADDL avant de pouvoir les utiliser dans les modèles. Ces deux phases sont consécutives et itératives et assurent que SADDL décrit précisément et de manière structurée des défauts de conception en accord avec les fiches de règles. Aussi, nous avons appliqué notre méthodologie sur une quinzaine de défauts de conception dont une dizaine d'anti-patrons définis par Brown *et al.* [BRO 98] et Dudney *et al.* [DUD 03] ainsi que quelques mauvaises odeurs et défauts de patrons.

5.1. *Les défauts de patron*

Les défauts de patron sont représentés de la même façon que les patrons de conception classiques mais avec des contraintes supplémentaires sur les relations entre les entités et les éléments. Par exemple, un défaut du patron de conception Observer peut être la fusion des rôles *Sujet* et *SujetConcret* (*cf.* figure 9). Ce défaut de patron est un patron déformé dû au choix d'avoir une classe jouant deux rôles.

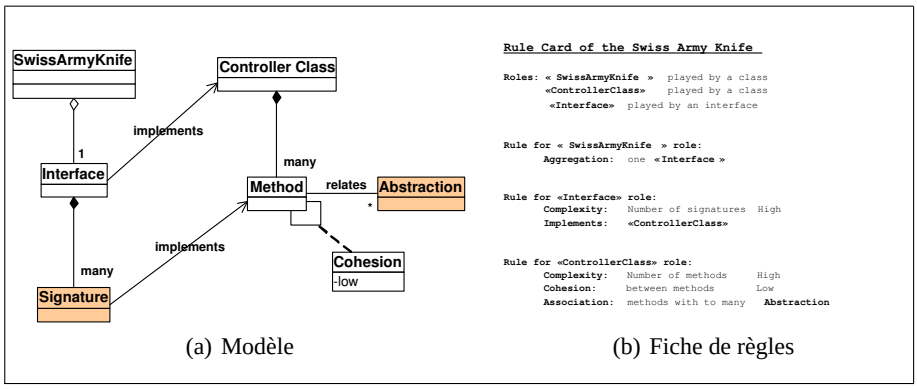


(a) Patron Observer (b) Patron déformé

Figure 9. *Le patron Observer et son patron déformé*

5.2. *Les anti-patrons*

Nous avons instancié les anti-patrons représentés dans la figure 5. Un modèle d'anti-patron comme le Swiss Army Knife ou le Blob prend la forme d'une instance de la classe `AntiPattern` ou de l'une de ses sous-classes. Par exemple, le Swiss Army Knife [BRO 98] est une classe complexe qui offre un grand nombre de services représentés dans le modèle SADDL par le concept d'abstraction (*cf.* figure 10). Une classe utilitaire est un bon exemple de couteau suisse.



(a) Modèle (b) Fiche de règles

Figure 10. *Le couteau suisse*

5.3. *Symptômes*

De la même façon que nous avons spécifié les défauts de patron et les anti-patrons, nous avons également décrit des symptômes assez complexes, tels que le Shotgun Surgery et le Divergent Change (*cf.* figure 11). Une classe dont un changement altère de nombreuses autres classes, à cause d'un couplage fort entre ces classes, est symptomatique d'un Shotgun Surgery. Une classe modifiée de différentes façons pour différentes raisons est symptomatique d'un Divergent Change.

Blob						
	ArgoUML	Azureus	GanttProject	PMD	QuickUML	Total
Effectifs	3	0	1	0	1	5
Precisions	3/3 = 100%	0/0 = 100%	2/1 !	0/0 = 100%	0/1 = 0%	5 = 100%
Swiss Army Knife						
	ArgoUML	Azureus	GanttProject	PMD	QuickUML	Total
Effectifs	14	143	21	5	3	186
Precisions	3/14 = 21.42%	63/143 = 44.05%	9/21 = 42.84%	2/5 = 40%	0/3 = 0%	77/186 = 41.39%

Tableau 1. *Résultats des algorithmes de détection.*

5.4. *Détection des défauts de conception*

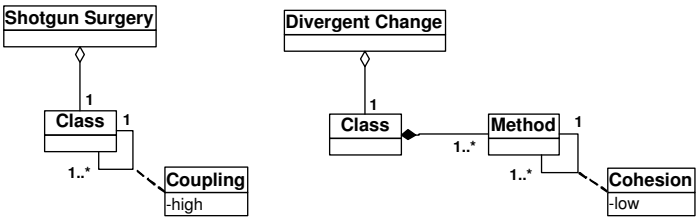
Suite à la description et la modélisation des défauts de conception, nous avons pu inférer semi-automatiquement des algorithmes de détection. Le tableau 1 fournit de premiers résultats de la détection de deux défauts de conception (Blob, Swiss Army Knife) sur sur 5 logiciels libres (ArgoUML, Azureus, GanttProject, PMD, QuickUML). Nos algorithmes de détection ont eu, en général, une précision plus grande ou égale à 40%. Les résultats montrent la précision de la détection et ainsi l'utilité de notre méthodologie et de la description des défauts de conception.

5.5. *Discussions*

Les phases de modélisation et validation se sont faites en parallèle par deux des auteurs. Des modèles ont été conjointement comparés, corrigés et affinés selon notre méthodologie itérative afin d'aboutir à des modèles précis et fidèles aux fiches de règles. À l'issue des phases de modélisation et validation, nous avons enrichi notre métamodèle d'autres concepts qui n'avaient pas été définis. La phase de validation s'est avérée primordiale dans notre méthodologie car elle a permis d'améliorer l'analyse des concepts clefs, leur taxonomie, le métamodèle et leur modélisation. La métamodélisation et la modélisation sont encore parfois limitées ; néanmoins, il s'agit d'un point de départ intéressant pour des débats constructifs, d'autant que notre méthodologie a permis de spécifier une quinzaine de défauts de conception.

6. Conclusion

La détection et la correction des défauts de conception représentent des enjeux importants pour améliorer la qualité des programmes, faciliter leur évolution et leur maintenance et, ainsi, épargner des ressources précieuses. Cependant, avant de pouvoir détecter ou corriger les défauts de conception, il faut en posséder des représentations précises et structurées. Le manque de travaux sur la spécification des défauts nous a amené à présenter une taxonomie et un métamodèle, pour préciser leurs descriptions textuelles et confuses et les heuristiques existantes. Nous proposons une taxonomie



(a) Shotgun Surgery (b) Divergent Change

Figure 11. *Les modèles du Shotgun Surgery et du Divergent Change*

et un métamodèle des défauts de conception pour : (1) faciliter la compréhension et l'identification des défauts de conception grâce à la terminologie et la classification des défauts ; (2) fournir un catalogue de modèles de défauts dans une forme succincte et facilement compréhensible comme il existe pour les patrons de conception ; (3) évaluer et améliorer la qualité des programmes en localisant les défauts ; (4) fournir un cadre de développement de techniques et d'outils pour la détection et la correction des défauts. Dans nos prochains travaux, nous comptons d'abord affiner la description des règles, augmenter notre base de défauts puis développer nos implantations de différentes techniques de détection.

Remerciements. Les auteurs remercient Jean-François Djoufak pour ses commentaires sur des versions préliminaires de ce travail ainsi que les relecteurs anonymes.

7. Bibliographie

[ALB 02] ALBIN-AMIOT H., COINTE P., GUÉHÉNEUC Y.-G., « Un méta-modèle pour coupler application et détection des design patterns », DAO M., HUCHARD M., Eds., *actes du 8^e colloque Langages et Modèles à Objets*, vol. 8, numéro 1-2/2002 de *RSTI – L'objet*, Hermès Science Publications, janvier 2002, p. 41–58.

[BOR 05] BORODAY S., PETRENKO A., SINGH J., HALLAL H., « Dynamic analysis of java applications for multithreaded antipatterns », *WODA '05 : Proceedings of the third international workshop on Dynamic analysis*, New York, NY, USA, 2005, ACM Press, p. 1-7.

[BRO 98] BROWN W. J., MALVEAU R. C., BROWN W. H., III H. W. M., MOWBRAY T. J., *Anti Patterns : Refactoring Software, Architectures, and Projects in Crisis*, John Wiley and Sons, 1^{st} édition, March 1998.

[DEM 02] DEMEYER S., DUCASSE S., NIERSTRASZ O., *Object-Oriented Reengineering Patterns*, Morgan Kaufmann, 2002.

[DUD 03] DUDNEY B., ASBURY S., KROZAK J., WITTKOPF K., *J2EE AntiPatterns*, Wiley, 2003.

[FOW 99] FOWLER M., *Refactoring – Improving the Design of Existing Code*, Addison-Wesley, 1^{st} édition, June 1999.

[GAM 94] GAMMA E., HELM R., JOHNSON R., VLISSIDES J., *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1^{st} édition, 1994.

[GUÉ 01] GUÉHÉNEUC Y.-G., ALBIN-AMIOT H., « Using Design Patterns and Constraints to Automate the Detection and Correction of Inter-Class Design Defects », LI Q., RIEHLE R., POUR G., MEYER B., Eds., *proceedings of the* 39^{th} *conference on the Technology of Object-Oriented Languages and Systems*, IEEE Computer Society Press, July 2001, p. 296–305.

[LAN 06] LANZA M., MARINESCU R., DUCASSE S., *Object-oriented Metrics in Practice*, Springer-Verlag, 2006, To appear.

[MAN 03] MANTYLA M., « Bad Smells in Software - a Taxonomy and an Empirical Study. », PhD thesis, Helsinki University of Technology, 2003.

[MOH 05a] MOHA N., « catalogue of Design Defects », rapport, October 2005, Department of Computer Science and Operations Research, University of Montréal, URL : `http://ptidej.iro.umontreal.ca/Members/mohanaou/technicalreport/CatalogDesignDefects.pdf`.

[MOH 05b] MOHA N., GUÉHÉNEUC Y.-G., « On the Automatic Detection and Correction of Software Architectural Defects in Object-Oriented Designs », DEMEYER S., MENS K., WUYTS R., DUCASSE S., Eds., *Proceedings of the* 6^{th} *ECOOP Workshop on Object-Oriented Reengineering*, July 2005.

[MOH 05c] MOHA N., HUYNH D.-L., GUÉHÉNEUC Y.-G., « A Taxonomy and a First Study of Design Pattern Defects », ANTONIOL G., GUÉHÉNEUC Y.-G., Eds., *Proceedings of the STEP International Workshop on Design Pattern Theory and Practice (IWDPTP05)*, September 2005.

[MUN 05] MUNRO M. J., « Product Metrics for Automatic Identification of “Bad Smell" Design Problems in Java Source-Code », LANUBILE F., SEAMAN C., Eds., *proceedings of the* 11^{th} *International Software Metrics Symposium*, IEEE Computer Society Press, September 2005.

[RIE 96] RIEL A. J., *Object-Oriented Design Heuristics*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[SMI 02] SMITH C. U., WILLIAMS L. G., *Performance Solutions : A Practical Guide to Creating Responsive, Scalable Software*, Addison-Wesley Professional, Boston, MA, USA, 2002.

[TAT 02] TATE B. A., FLOWERS B. R., *Bitter Java*, Manning Publications, 2002.

[TIC 00] TICHELAAR S., DUCASSE S., DEMEYER S., NIERSTRASZ O., « A Meta-model for Language-Independent Refactoring », *Proceedings of ISPSE '00 (International Conference on Software Evolution)*, IEEE Computer Society Press, 2000, p. 157-167.

[TRI 03] TRIFU A., DRAGOS I., « Strategy-Based Elimination of Design Flaws in Object-Oriented Systems », DEMEYER S., DUCASSE S., MENS K., Eds., *proceedings of the* 4^{th} *international Workshop on Object-Oriented Reengineering*, Universiteit Antwerpen, July 2003.

[WAK 03] WAKE W. C., *Refactoring Workbook*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[WEN 01] WENDORFF P., « Assessment of Design Patterns During Software Reengineering : Lessons Learned from a Large Commercial Project », SOUSA P., EBERT J., Eds., *proceedings of* 5^{th} *Conference on Software Maintenance and Reengineering*, IEEE Computer Society Press, March 2001, p. 77–84.