

Répertoire des défauts de conception

Alban TIBERGHIEEN - Naouel MOHA - Tom MENS - Kim MENS

November 21th, 2007

Technical Report 1303, University of Montreal

Contents

1	The Bloaters	3
1.1	Long Method	3
1.2	Large Class	3
1.3	Primitive Obsession	4
1.4	Long Parameter List	4
1.5	Data Clumps	4
2	The Object-Oriented Abusers	5
2.1	Switch Statement	5
2.2	Temporary Field	5
2.3	Refused Bequest	6
2.4	Tradition Breaker	6
2.5	Alternative Classes with Different Interfaces	6
2.6	Access to Children	6
2.7	Poor Usage of Abstract Classes	7
2.8	Not Enough Information Hiding	7
2.9	Too Much Information Hiding	8
2.10	Poor Usage of Interfaces	8
3	The Dispensables	8
3.1	Lazy Class	8
3.2	Data Class	9
3.3	Duplicated Code	9
3.4	Dead Code	9
3.5	Speculative Generality	9

4	The Change Preventers	10
4.1	Divergent Change	10
4.2	Shotgun Surgery	10
4.3	Parallel Inheritance Hierarchies	10
5	The Couplers	10
5.1	Feature Envy	10
5.2	Inappropriate Intimacy	11
5.3	Message Chains	11
5.4	Middle Man	11
5.5	Group of Interdependent Objects	12
5.6	Circular Dependencies	12
6	Others	12
6.1	Incomplete Library Class	12
6.2	Too Much Comments	12
6.3	Not Enough Comments	13
7	Anti-patrons	13
7.1	The Blob	13
7.2	Functional Decomposition	13
7.3	Spaghetti Code	14
7.4	Swiss Army Knife	14
8	Code Level Defects	14
8.1	Copying / Assignment / Initialisation	14
8.2	Equality and Identity Testing	15
8.3	Problematic Conditionals	15
8.4	Pathological Switch	16
8.5	Constructor Problems	16
8.6	Instantiation Problems	17
8.7	Memory Management Problems	17
8.8	Exception Handling Problems	17
8.9	Concurrency Problems	18
8.10	Scoping Problems	18
8.11	Unrespected Naming Rules	18

Afin de classer un certain nombre de défauts de conception, nous avons utilisé la classification proposée par Mäntylä [2] qui permet de mieux comprendre les différents défauts et les relations qui les lient. Nous avons décrit chaque défaut en suivant le patron suivant: le (ou les) nom(s) connu(s) de ce défaut, un identificateur (ID), une description, et dans certains cas des exemples.

1 The Bloaters

Cette famille de code smells caractérise des entités devenues si grande qu’elles sont devenues difficiles à gérer.

1.1 Long Method

Nom(s) : Long Method [1, 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. Dans nos expériences, nous considérons qu’une méthode longue a plus de 35 instructions et de très longue une méthode avec plus de 90 instructions. Il est à noter qu’il s’agit de valeurs seuils et donc des valeurs arbitraires: une méthode contenant 34 attributs peut également être considérée comme longue mais avec un degré de vérité de 95% par exemple¹. 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.

1.2 Large Class

Nom(s) : Large Class [1, p. 78]

ID : `largeClass`

¹Nous avons obtenu ce chiffre en analysant une dizaine de programmes. Nous avons mesuré le nombre d’instructions de toutes les méthodes de toutes les classes de chacun des programmes et nous avons appliqué des algorithmes de clustering pour extraire 3 clusters correspondant aux 3 catégories différentes de méthodes: les petites, les moyennes et les grandes. L’analyse de ces données nous a permis de récupérer les valeurs seuils associées à chacune des catégories.

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. Nous considérons qu'une classe est large si elle a plus de 76 attributs et/ou méthodes et de très large une classe avec plus de 183 attributs et/ou méthodes. Il est à noter qu'il s'agit de valeurs seuils et donc des valeurs arbitraires: une classe contenant 75 attributs peut également être considérée comme large mais avec un degré de vérité de 95% par exemple.

1.3 Primitive Obsession

Nom(s) : Primitive Obsession [1, p. 81]

ID : `primitiveObsession`

Description : L'utilisation abusive des primitives tels 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.

1.4 Long Parameter List

Nom(s) : Long Parameter List [1, 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. Nous considérons qu'à partir de 4 paramètres, il s'agit d'une longue liste de paramètres.

1.5 Data Clumps

Nom(s) : Data Clumps [1, p. 81]

ID : `dataClumps`

Description : Il s'agit d'un ensemble de variables fortement cohésives. 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 3 entiers pour les couleurs RGB.

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.

2.1 Switch Statement

Nom(s) : Switch Statement [1, 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 Temporary Field

Nom(s) : Temporary Field [1, p. 84]

ID : `temporaryField`

Description : Certains objets contiennent des attributs qui ne sont pas utilisés dans tous les cas. Ces attributs 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. 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.

2.3 Refused Bequest

Nom(s) : Refused Bequest [1, 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.

2.4 Tradition Breaker

Nom(s) : Tradition Breaker [3]

ID : `traditionBreaker`

Description : Ce défaut désigne une classe héritée qui brise la 'tradition' en fournissant un large ensemble de services qui n'ont pas de liens avec les services héritées par la classe mère.

2.5 Alternative Classes with Different Interfaces

Nom(s) : Alternative Classes with Different Interfaces [1, 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.

2.6 Access to Children

Nom(s) : Access to Children

ID : `accessToChildren`

Description : Ce défaut désigne le cas où une classe utilise directement l'une de ses sous-classes. Ce défaut se présente quand une classe accède directement à une ou plusieurs des ses sous-classes. En principe ceci doit être évité car une classe n'est pas censée connaître ses sous-classes.

2.7 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.

2.8 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'information 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'information.

2.9 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 privées.

2.10 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.

3 The Dispensables

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

3.1 Lazy Class

Nom(s) : Lazy Class [1, p. 83]

ID : `lazyClass`

Description : Ce sont des classes qui n'ont pas une grande utilité car elle 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.

3.2 Data Class

Nom(s) : Data Class [1, p. 86]

ID : `dataClass`

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

3.3 Duplicated Code

Nom(s) : Duplicated Code [1, p. 76]

ID : `duplicatedCode`

Description : Ce défaut apparaît dans le code soit de manière explicite soit de manière subtile. La forme explicite correspond à du code redondant, alors que la forme subtile correspond à du code qui fait la même chose mais utilisant des combinaisons de données ou de comportements différents.

3.4 Dead Code

Nom(s) : Dead Code / Unused Code / Unnecessary Code

ID : `deadCode`

Description : Ce défaut désigne du code 'mort', inutilisé ou inutile. Il s'agit généralement de code qui n'est pas utilisé et dont personne n'ose supprimer car personne ne se souvient à quoi il sert.

3.5 Speculative Generality

Nom(s) : Speculative Generality [1, 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.

4 The Change Preventers

Cette catégorie de défauts désigne les défauts ou structures de code qui rendent difficile la modification du logiciel.

4.1 Divergent Change

Nom(s) : Divergent Change [1, p. 79]

ID : `divergentChange`

Description : Ce défaut désigne le fait qu'une classe donnée soit modifiée de différentes façons pour différentes raisons.

4.2 Shotgun Surgery

Nom(s) : Shotgun Surgery [1, p. 80] / High Coupling

ID : `shotgunSurgery`

Description : Ce défaut désigne le fait que l'on soit obligé de modifier plusieurs classes lorsque l'on désire faire un changement donné.

4.3 Parallel Inheritance Hierarchies

Nom(s) : Parallel Inheritance Hierarchies [1, p. 83]

ID : `parallelInheritanceHierarchies`

Description : Ce défaut est un cas spécial du Shotgun Surgery. Il désigne le fait que lorsqu'on crée une sous-classe d'une classe, on soit obligé de créer une sous-classe d'une autre classe.

5 The Couplers

Cette catégorie désigne les défauts qui sont fortement couplés.

5.1 Feature Envy

Nom(s) : Feature Envy [1, p. 80]

ID : `featureEnvy`

Description : Ce défaut désigne le fait qu'une méthode fait beaucoup d'appels à d'autres classes afin d'obtenir des données et des fonctionnalités.

Exemple(s) : il peut s'agir d'une méthode qui invoque de nombreuses méthodes `get` d'une autre classe afin de pouvoir réaliser une opération.

5.2 Inappropriate Intimacy

Nom(s) : Inappropriate Intimacy [1, p. 85]

ID : `inappropriateIntimacy`

Description : Il s'agit de classes qui passent trop de temps à fouiller dans d'autres classes et donc entrent trop dans l'intimité de la classe. Plus spécifiquement, il s'agit de deux classes qui sont très fortement liées entre elles.

5.3 Message Chains

Nom(s) : Message Chains [1, 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()`.

5.4 Middle Man

Nom(s) : Middle Man [1, p. 85]

ID : `middleMan`

Description : Ce défaut désigne un abus du concept de délégation. Plus précisément, il s'agit d'une classe qui délègue trop de méthodes à une autre classe. La classe ne fait alors que passer des méthodes à d'autres classes.

5.5 Group of Interdependent Objects

Nom(s) : Group of Interdependent Objects

ID : `interdependentObjects`

Description : Ce défaut désigne une classe qui dépend immédiatement de beaucoup d'autres classes et d'autres classes dépendent immédiatement de celle-ci.

5.6 Circular Dependencies

Nom(s) : Circular Dependencies

ID : `circularDependencies`

Description : Une dépendance circulaire est lorsqu'on commence à partir d'une classe donnée A, on peut revenir sur cette même classe en suivant les liens de dépendance.

6 Others

Cette catégorie regroupe les défauts qui n'ont pas pu être classés dans l'une des catégories de cette classification.

6.1 Incomplete Library Class

Nom(s) : Incomplete Library Class [1, p. 86]

ID : `incompleteLibraryClass`

Description : Ce défaut désigne les responsabilités qui émergent du code et qui devraient être déplacées dans une classe de librairie. Mais, il semble difficile d'ajouter ces nouvelles responsabilités dans la classe de librairie.

6.2 Too Much Comments

Nom(s) : Too Much Comments

ID : `tooMuchComments`

Description : Dans certains cas, l'utilisation abusive de commentaires dans le code sont là pour expliquer du mauvais code.

6.3 Not Enough Comments

Nom(s) : Not Enough Comments

ID : notEnoughComments

Description : Le manque de commentaires dénote une négligence au niveau de la documentation du programme.

7 Anti-patterns

Les anti-patterns sont des défauts de conception plus complexes qui reposent sur l'existence de code smells.

7.1 The Blob

Nom(s) : Blob [4, p. 73]

ID : blob

Description : Le Blob est une classe centralisant le traitement et donc par conséquent 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 procédurale). Ces classes sont difficiles à réutiliser et à tester. De plus, elles utilisent beaucoup de ressources. Cet anti-pattern 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.

7.2 Functional Decomposition

Nom(s) : Functional Decomposition [4, p. 97] / Module Mimic

ID : functionalDecomposition

Description : On observe cet anti-pattern lorsqu'un programmeur, à l'aise avec un langage procédural, transforme ces 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.

7.3 Spaghetti Code

Nom(s) : Spaghetti Code [4, 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.

7.4 Swiss Army Knife

Nom(s) : Swiss Army Knife [4, 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.

8 Code Level Defects

Les défauts au niveau code sont des problèmes à un très fin niveau de granularité qui sont localisés au niveau des lignes de code et qui peuvent découler d'une mauvaise façon d'utiliser des constructions de langage de programmation.

8.1 Copying / Assignment / Initialisation

Nom(s) : Copying / Assignment / Initialisation

ID : copyingAssignmentInitialisation

Description : Cette catégorie comprend tous les défauts au niveau code liés à un mauvais usage des initialisations, des affectations et des copies de variables. Il est difficile de donner une liste exhaustive des problèmes de cette catégorie car les différents outils de détection considèrent des types de problèmes de cette catégorie bien différents.

Exemple(s) :

- l'affectation d'une variable qui n'a pas d'effet car la variable n'est pas utilisée dans la suite du programme ou car elle est immédiatement assignée à une autre valeur;
- l'utilisation de variables non initialisées;
- l'affectation de variables à null;
- la manipulation d'une copie de l'objet au lieu de l'objet lui-même (et inversement);
- une copie superficielle de l'objet a été utilisée au lieu d'une copie intégrale (et inversement);
- des champs immuables.
- ...

8.2 Equality and Identity Testing

Nom(s) : Equality and Identity Testing

ID : equalityIdentityTesting

Description : Cette catégorie comprend tous les défauts de niveau code qui sont liés à l'identité et l'égalité entre objets.

8.3 Problematic Conditionals

Nom(s) : Problematic Conditionals

ID : problematicConditionals

Description : Cette catégorie fait référence aux conditions logiques du type (if ...else ...) qui sont compliquées inutilement.

Exemple(s) :

- Instructions `if` imbriquées

- Instructions conditionnelles sous la forme négative (if not ... else ...)
- ...

8.4 Pathological Switch

Nom(s) : Pathological Switch

ID : pathologicalSwitch

Description : Cette catégorie est liée à la précédente, mais fait référence à une mauvaise utilisation des instructions `switch`.

Exemple(s) :

- instruction `switch` sans `default`;
- instruction `default` vide;
- ...

Attention, ce défaut ne doit pas être confondu avec le défaut *Switch Statement* dans la catégorie *The Object-Oriented Abusers*.

8.5 Constuctor Problems

Nom(s) : Constuctor Problems

ID : constuctorProblems

Description : Cette catégorie fait référence aux défauts liés à un mauvais usage des constructeurs en Java.

Exemple(s) :

- passer les constructeurs à `null`;
- trop de constructeurs.

8.6 Instantiation Problems

Nom(s) : Instantiation Problems

ID : instantiationProblems

Description : Cette catégorie fait référence aux problèmes liés à l'instanciation des classes, c.-à-d. la création des objets sous forme d'instances de classes.

Exemple(s) :

- des classes publiques avec aucun constructeur qui peut être instancié à l'extérieur;
- des classes feuilles qui n'ont pas d'instances semblent ne pas être utilisées.

8.7 Memory Management Problems

Nom(s) : Memory Management Problems

ID : memoryManagementProblems

Description : Cette catégorie fait référence aux défauts liés à la gestion de la mémoire.

Exemple(s) : les problèmes liés à l'utilisation des `finalizer` en Java.

8.8 Exception Handling Problems

Nom(s) : Exception Handling Problems

ID : exceptionHandlingProblems

Description : Le mécanisme de gestion des exceptions de Java (`try`, `catch`, `throw`) est aussi une source commune de problèmes.

Exemple(s) :

- une instruction `return` dans un bloc `finally`;
- une instruction `throw` dans un bloc `finally`;
- des blocs `catch` vides;
- des blocs `finally` vides;
- des blocs au niveau des `catch` et des `throw`;
- des pauvres propagations d'exception.
- ...

8.9 Concurrency Problems

Nom(s) : Concurrency Problems

ID : `concurrencyProblems`

Description : Cette catégorie regroupe les problèmes liés au mécanisme de concurrence utilisé en Java tels que les threads.

8.10 Scoping Problems

Nom(s) : Scoping Problems

ID : `scopingProblems`

Description : Cette catégorie décrit les problèmes liés à la portée des variables et des méthodes au sein d'une classe.

Exemple(s) :

- des variables locales qui masquent les attributs de la classe;
- les paramètres des méthodes qui ont les mêmes noms que les attributs de la class';
- des déclarations de champs qui masquent des champs qui sont accessibles dans un super type.

8.11 Unrespected Naming Rules

Nom(s) : Unrespected Naming Rules

ID : `namingRules`

Description : Cette catégorie regroupe tous les défauts liés aux règles de nommage.

Exemple(s) :

- des méthodes ou des classes avec de long noms;
- une méthode qui a le même nom qu'un constructeur;
- des noms d'identificateurs vagues;
- des variables qui ont des noms identiques à ceux de types.

References

- [1] Martin Fowler. *Refactoring – Improving the Design of Existing Code*. Addison-Wesley, 1st edition, June 1999.
- [2] 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*, page 381, Washington, DC, USA, 2003. IEEE Computer Society. <http://www.soberit.hut.fi/mmantyla/BadCodeSmellsTaxonomy.htm>.
- [3] Radu Marinescu Michele Lanza. *Object-oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-oriented Systems*. Springer, October 2006.
- [4] Hays W. McCormick III Thomas J. Mowbray John Wiley & Sons Inc. William J. Brown, Raphael C. Malveau. *AntiPatterns – Refactoring Software, Architectures, and Projects in Crisis*. Robert Ipsen, 1998.