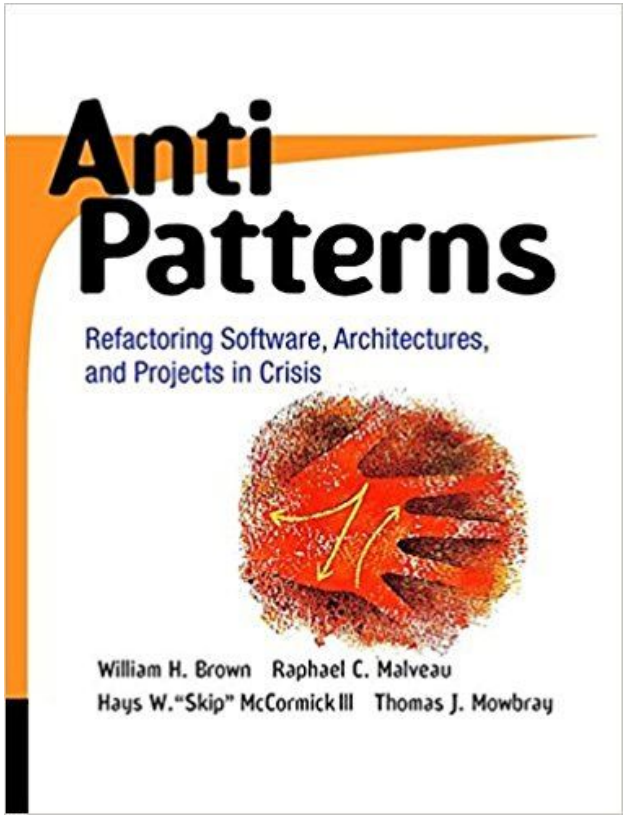
LOG8371 : Ingénierie de la qualité en logiciel

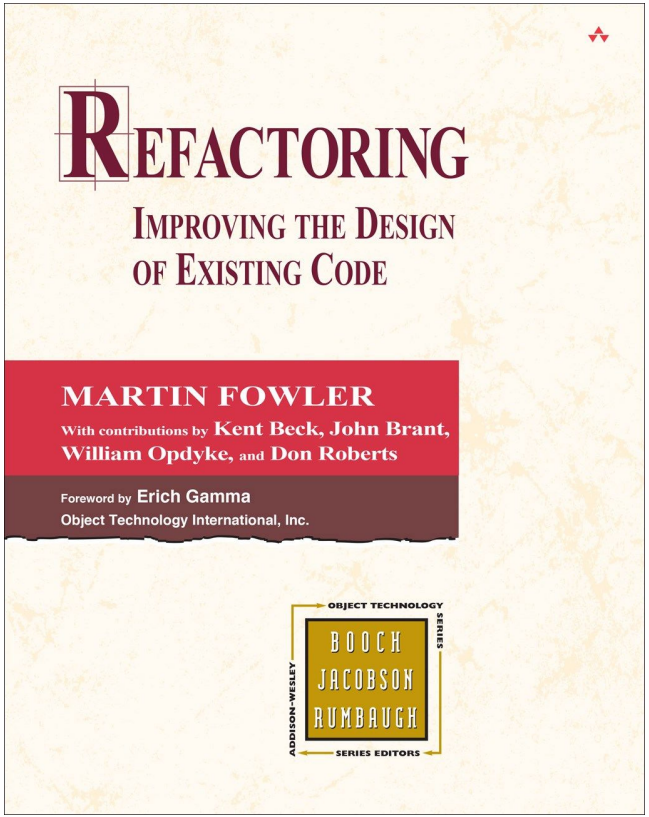
Qualité du Code - Anti-patrons

Hiver 2017

Fabio Petrillo
Chargé de Cours

POLYTECHNIQUE MONTRÉAL
LE GÉNIE EN PREMIÈRE CLASSE


Anti
Patterns
Refactoring Software, Architectures,
and Projects in Crisis
William H. Brown Raphael C. Malveau
Hays W. "Skip" McCormick III Thomas J. Mowbray


REFACTORING
IMPROVING THE DESIGN
OF EXISTING CODE
MARTIN FOWLER
With contributions by Kent Beck, John Brant,
William Opdyke, and Don Roberts
Foreword by Erich Gamma
Object Technology International, Inc.
OBJECT TECHNOLOGY SERIES
BOOCH
JACOBSON
RUMBAUGH
SERIES EDITORS
ADDISON-WESLEY

*“**Anti-patrons** Sont des choix de **conception médiocres** qui sont conjecturés pour que les systèmes orientés objet soient plus **difficiles à maintenir** ".*

Anti-patrons (Ou code odeur)

- Un Anti-patron est une **forme littéraire** qui décrit une solution commune à **un problème** qui génère des **conséquences négatives**.
- Les anti-patrons offrent une **expérience** réelle dans la reconnaissance des **problèmes récurrents** dans l'industrie du logiciel.
- Les anti-patrons offrent un **vocabulaire commun** pour identifier les problèmes et discuter des solutions.
- Plus de 30 antipatterns ont été catalogués, mais nous nous concentrerons sur les 12 plus discutés par le milieu universitaire et il a des outils de détection automatisés.

Étude de l'impact des anti-patrons

- Les classes qui participent à des anti-patrons sont plus susceptibles de **changer et d'inclure les erreurs** que d'autres.
- Les classes **étendues** participant aux anti-patrons ont des chances **plus élevées de changer** ou d'être sujettes à **la correction des erreurs** que les autres classes.
- **La taille seule** ne peut pas expliquer la probabilité plus élevée de classes avec anti-patrons de subir un changement (de correction de faute) que d'autres.
- Les changements structurels **affectent plus** de classes avec anti-patrons que d'autres.
- Nous avons maintenant des **preuves empiriques** que le code contenant des code smells ou qui participe à des anti-patrons est beaucoup plus propice aux changements que le code «propre».

Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc and Guiliano Antoniol, An Exploratory Study of the Impact of Antipatterns on Class Change- and Fault-Proneness, Journal of Empirical Software Engineering (EMSE), 2011.

Anti-patrons/code smells - résultats de recherche

- Les **nouveaux arrivants** ne sont **PAS responsables** de l'introduction de mauvais smalls, tandis que les développeurs avec **des charges de travail élevées et la pression de libération** sont plus enclins à introduire des instances smell.
- Les résultats ont montré que la plupart des instances smell sont introduites **lorsqu'un artefact est créé et pas suite à son évolution.**
- 80% de smells survivent dans le système. Code smells ont une **grande capacité de survie** et sont **rarement éliminées** en conséquence directe des activités de **refactorisation**.
- Parmi les 20% des instances supprimées, seulement 9% sont supprimés en conséquence directe des opérations de refactorisation.

M. Tufano; F. Palomba; G. Bavota; R. Oliveto; M. Di Penta; A. De Lucia; D. Poshyvanyk, "When and Why Your Code Starts to Smell Bad (and Whether the Smells Go Away)," in IEEE Transactions on Software Engineering , vol.PP, no.99, pp.1-1

Anti-patrons - résultats de recherche

- Tous ces résultats suggèrent que les codes smells et les anti-patrons **doivent être soigneusement détectés et surveillés** et, chaque fois que cela est nécessaire, les opérations de **refactorisation** devraient être **planifiées et exécutées** pour les traiter.
- L'**identification** et la **correction** des anti-patrons / code smells dans des systèmes logiciels volumineux et non triviaux peuvent être **très difficiles**.
- Les anti-patrons peuvent être **définis** en termes de seuils imposés sur les **valeurs métriques** (Moha et al., 2008).

Fabio Palomba, Andrea De Lucia, Gabriele Bavota, Rocco Oliveto: Anti-Pattern Detection: Methods, Challenges, and Open Issues. Advances in Computers 95: 201-238 (2015)

Anti-patrons

- Plus de 30 anti-patrons ont été catalogués par Fowler et Brown, mais nous nous concentrerons sur les 12 plus discutés par le milieu universitaire, fournissant des **outils de détection automatisés**.
  - Blob or God Class, Feature Envy
  - Duplicate Code, Refused Bequest
  - Divergent Change, Shotgun Surgery
  - Parallel Inheritance, Functional Decomposition
  - Spaghetti Code, Swiss Army Knife
  - Type Checking, Poltergeists

Blob or God Class

- Une classe ayant **une taille énorme** et mettant en œuvre différentes **responsabilités.**
- Blob est une classe qui **centralise** la plupart des comportements du système.
- Présence d'un **nombre élevé d'attributs et de méthodes**, qui implémentent **différentes fonctionnalités** et par de nombreuses **dépendances** avec des classes de **données**.
- “Selfish behavior” parce qu'il fonctionne pour lui-même.
- Stratégies de détection
  - LCOM (Lack of Cohesion Of Methods) higher than 20
  - Number of methods and attributes higher than 20
  - It has an one-to-many association with data classes
  - Name that contains a suffix in the set {*Process, Control, Command, Manage, Drive, System*}

Feature Envy

- Une méthode faisant **trop d'appels** vers des méthodes d'une **autre classe** pour obtenir des données et / ou des fonctionnalités.
- Il se caractérise souvent par un **grand nombre de dépendances** avec la classe enviée.
- Une influence **négative** sur la **cohésion et le couplage** de la classe dans laquelle la méthode est mise en œuvre.
- Stratégies de détection
    - Parcourir Abstract Syntax Tree (AST) d'un système logiciel afin d'identifier, pour chaque champ, l'ensemble des classes de référencement.
    - Définissez un seuil pour discriminer les champs ayant trop de références avec d'autres classes.

Duplicate Code

- Des classes qui montrent la **même structure de code** dans plusieurs endroit.
- La duplication de code est un problème potentiellement grave qui **affecte** la **maintenabilité** d'un système logiciel, mais aussi sa **compréhension**.
- Le problème de **l'identification** de la duplication de code est **très difficile** car, lors de l'évolution, **différentes copies** d'une caractéristique subissent des changements différents, ce qui affecte la possibilité d'identifier les fonctionnalités communes fournies par différentes fonctionnalités copiées.
- Stratégies de détection
  - Détection de la similitude syntaxique ou structurelle du code source
  - Pour trouver des correspondances de sous-arbres (en utilisant AST)

Refused Bequest

- Une classe **héritant** de fonctionnalités qu'il n'**utilise jamais**.
- Une sous-classe **remplace beaucoup de méthodes** héritées par leurs parents.
- Relation d'**héritage est erroné**, à savoir la sous-classe n'est pas une spécialisation du parent..
- Les méthodes ne sont **jamais appelées** par les clients de la sous-classe
- Stratégies de détection
  - La classe remplace plus de x% des méthodes définies par le parent.
  - Combinaison d'analyse de code source statique et d'exécution de test d'unité dynamique.
  - remplacer intentionnellement ces méthodes introduisent une erreur dans la nouvelle implémentation (par exemple, la division par zéro) dans la super-classe. Si l'erreur ne vient jamais, la sous-classe est Refused Bequest antipattern.
  - Difficile à détecter automatiquement.

Changement Divergent

- Une classe a souvent changé de **différentes façons** pour différentes raisons.
- Les classes affectées par cet anti-modèle ont généralement une **faible cohésion**.
- Stratégies de détection
  - En utilisant des informations de couplage, il est possible de construire une matrice de Design Change Propagation Probability (DCPP). Le DCPP est une matrice n x n où l'entrée générique Aij est la probabilité qu'un changement de conception sur l'artefact i nécessite un changement également à l'artefact j.
  - En utilisant les informations historiques qu'un système peut avoir (i.e., change log)
  - Détecter un sous-ensemble de méthodes dans la même classe qui changent souvent ensemble

Shotgun Surgery

- Une classe où une modification implique des **changements en cascade** dans plusieurs classes liées.
- Chaque fois que vous faites une sorte de changement, vous devez apporter **beaucoup de changements** à beaucoup de cours différents.
- Eg: Changement dans UI (View) -> Controller -> Model -> Persistence
- Stratégies de détection
  - En utilisant des informations de couplage, il est possible de construire une matrice de Design Change Propagation Probability (DCPP). Le DCPP est une matrice n x n où l'entrée générique Aij est la probabilité qu'un changement de conception sur l'artefact i nécessite un changement également à l'artefact j.
  - En utilisant les informations historiques qu'un système peut avoir (i.e., change log)
  - Détecter un sous-ensemble de méthodes dans la même classe qui changent souvent ensemble

Héritage parallèle

- Paire de classes où l'**ajout d'une sous-classe** dans une hiérarchie implique l'ajout d'une **sous-classe dans une autre hiérarchie**.
- Chaque fois que vous faites une sous-classe d'une classe, vous devez également faire une sous-classe d'une autre.
- Un cas spécial de Shotgun Surgery
- Stratégie de détection
    - S'appuie sur des informations historiques

Décomposition fonctionnelle

- Une classe implémenter suivant un **style procédural**.
- Une classe dans laquelle l'héritage et le polymorphisme sont mal utilisés, déclarant de **nombreux champs privés et implémentant quelques méthodes**.
- Programmation de type procédural -> une **routine principale** qui **appelle** de nombreux **sous-programmes**
- Stratégies de détection
    - De nombreuses dépendances avec des classes composées d'un très petit nombre de méthodes portant sur une seule fonction

Spaghetti Code

- Une classe **sans structure** déclarant de **longues méthodes** sans paramètres.
- Caractérisé par des **méthodes complexes**, sans paramètres, interagissant entre eux à l'aide de variables d'instance.
- Programmation de type procédural. Il a généralement un **nom de procédure**.
- Stratégies de détection
  - À la recherche de classes ayant au moins une longue méthode
  - À savoir une méthode composée d'un grand nombre de LOC et déclarant aucun paramètre
  - La classe ne présente pas de caractéristiques de conception orientée objet
  - La classe n'utilise pas le concept d'héritage et doit utiliser de nombreuses variables globales

Swiss Army Knife

- Une classe qui présente une **grande complexité** et offre un **grand nombre** de **services** différents.
- Swiss Army Knife est une classe qui **répond** à un **large** éventail de **besoins**.
- Swiss Army Knife est une classe qui fournit des **services à d'autres classes**, **exposant une grande complexité**, tandis qu'une **Blob** est une classe qui **monopolise** le traitement et les données du système.
- L'anti-patron survient lorsqu'une classe possède de **nombreuses méthodes avec une grande complexité** et la classe possède un **nombre élevé d'interfaces**.
- Stratégies de détection
  - Métriques de complexité
  - Contrôles sémantiques afin d'identifier les différents services fournis par la classe

Type Checking

- Une classe qui montre des **déclarations conditionnelles compliquées**.
- Ce problème se manifeste surtout lorsqu'un développeur utilise des **instructions conditionnelles pour diffuser dynamiquement** le comportement du système au lieu du polymorphisme.
- Manque de connaissances sur la façon d'utiliser les mécanismes OO tels que le **polymorphisme**.
- Stratégies de détection
  - Voir des déclarations conditionnelles longues et compliquées
  - Une déclaration conditionnelle implique RunTime Type Identification (RTTI)

Poltergeists

- Poltergeists sont des classes avec des **responsabilités limitées** et des rôles à jouer dans le système.
- Poltergeists créent des **abstractions inutiles**; Ils sont excessivement complexes, difficiles à comprendre et difficile à entretenir.
- Attention aux contrôleurs: il peut être poltergeists.
- Stratégies de détection
  - Plusieurs parcours de navigation redondants
  - Classes apatrides
  - Objets et classes temporaires et de courte durée

Linguistic Anti-Patterns

*Linguistic Antipatterns (LAs) d*ans les systèmes logiciels, on constate des **pratiques médiocres dans la nomination**, la documentation et le **choix des identifiants** dans l'implémentation d'une entité, ce qui pourrait entraver la compréhension du programme.

V. Arnaoudova, M. Di Penta, G. Antoniol and Y. G. Guéhéneuc, "A New Family of Software Anti-patterns: Linguistic Anti-patterns," *2013 17th European Conference on Software Maintenance and Reengineering*, Genova, 2013, pp. 187-196.

“Get” - more than an accessor

Dans Java, les méthodes accessor, appelées aussi getters, fournir un moyen d'accéder aux attributs de classe. Cependant, **il n'est pas courant que les getters effectuent des actions autres que le renvoi de l'attribut correspondant**. Toute autre action devrait être documentée, éventuellement en nommant la méthode différemment que "getSomething".

“Get” - more than an accessor

```
public ImageData getImageData() {
  Point size = getSize();
  RGB black = new RGB(0, 0, 0);
  RGB[] rgbs = new RGB[256];
  rgbs[0] = black; // transparency
  rgbs[1] = black; // black
  PaletteData dataPalette=new PaletteData(rgbs);
  imageData = new
    ImageData(size.x,size.y,8,dataPalette);
  imageData.transparentPixel = 0;
  drawCompositeImage(size.x, size.y);
  for (int i = 0; i < rgbs.length; i++)
    if (rgbs[i] == null) rgbs[i] = black;
  return imageData;
}
```

Example of “Get” - more than an accessor (Eclipse-1.0).

“Is” returns more than a Boolean

Lorsqu'un nom de méthode commence par le terme "is" on s'attendrait à un type booléen comme retour, ayant ainsi deux valeurs possibles pour le prédicat, c'est-à-dire "vrai" et "faux". Ainsi, avoir une méthode "is" qui ne renvoie pas un Boolean, mais renvoie plus d'informations est counterintuitive.

```
public int isValid() {
 final long currentTime =
        System.currentTimeMillis();
 if (currentTime <= this.expires) {
  // The delay has not passed yet -
  // assuming source is valid.
  return SourceValidity.VALID;
 }
 // The delay has passed,
 // prepare for the next interval.
 this.expires = currentTime + this.delay;
 return this.delegate.isValid();
}
```

Example of “Is” returns more than a Boolean (Cocoon 2.2.0).

“Set” method returns

Une méthode définie ayant un type de retour différent de vide doit documenter les type/values de retour avec un commentaire approprié ou doit être nommé différemment pour éviter toute confusion.

```
public Dimension setBreadth
        (Dimension target, int source) {
  if (orientation == VERTICAL)
    return new Dimension(source,
        (int)target.getHeight());
  else
    return new Dimension(
                (int)target.getWidth(), source);
}
```

Example of “Set” method returns (ArgoUML-0.10.1).

Expecting but not getting a single instance

Lorsqu'un nom de méthode indique qu'un seul objet (et non une collection) est renvoyé, il doit être compatible avec son type de retour. Si, par contre, le type de retour est une collection, la méthode doit être renommée ou une documentation

```
/* Returns the expansion state for a tree.
 * @return the expansion state for a tree */
public List getExpansion() { return fExpansion;}
```

Example of Expecting but not getting a single instance (Eclipse-1.0)

Linguistic anti-patterns (cont.)

- **La méthode “Get” ne retourne pas**
  - protected **void get**MethodBodies(CompilationUnitDeclaration unit ...
- **Question pas répondue**: le nom de la méthode est sous la forme de prédicat alors que le type de retour n'est pas booléen.
  - public **void is**Valid (...
- **La méthode de transformation ne retourne pas**: Le nom de la méthode suggère la transformation d'un objet, mais il n'y a pas de valeur de retour.
  - public **void** java**To**Native (...
- **Expectant mais pas avoir une collection**: Le nom de la méthode suggère de renvoyer une collection, mais seulement un objet a retourné ou rien.
  - public **boolean** getStat**s** ( ) { return _stats ; }

Linguistic anti-patterns (cont.)

- **Annonce un mais contient beaucoup:** Un nom d'attribut suggère une instance unique, alors que son type suggère que l'attribut stocke une collection d'objets.
  - **Vector** _target;
- **Le nom suggère le boolean mais le type ne suggère pas:**
  - **int**[ ] **is**Reached;
- **Annonce plus qu'il ne contient**
  - private static **boolean** _stat**s** = **true**;

Antipattern detection tools

- God Class
  - http://pmd.sourceforge.net
- Duplication code
  - http://pmd.sourceforge.net/pmd-4.3.0/cpd.html
- SmellDetector (Ptidej - several smells)
  - git clone https://github.com/ptidej/SmellDetectionCaller.git
- Linguistic Anti-pattern Detector (LAPD)
  - http://www.veneraarnaoudova.ca/linguistic-anti-pattern-detector-lapd/
- JDeodorant (several smells)
  - https://github.com/tsantalis/JDeodorant

Les anti-patrons sont ils vraiment dangereux?

Malgré les preuves existantes sur les effets négatifs des anti-patrons, il n'est toujours **pas clair** si les développeurs considèrent réellement **tous les anti-patrons comme symptômes d'erreur de conception** / choix d'implémentation ou si certains d'entre eux ne sont qu'une **manifestation de la complexité intrinsèque** de la solution conçue.

Les études empiriques ont indiqué que (i) les développeurs ont perçu différentes instances de Blob comme n'étant pas **particulièrement dangereux** pour la maintenabilité du système, surtout parce qu'ils changent ces classes de manière sporadique; Et (ii) certains développeurs, en particulier les programmeurs juniors, fonctionnent mieux sur une version d'un système comportant certaines classes qui **centralisent le contrôle**, c'est-à-dire Blob.

La présence d'anti-patrons dans le code source est parfois **tolérable** et fait partie des choix de conception des développeurs.

Prochain cours
Logging, Debugging, Reviewing and Version Control System