

(OBJECT-ORIENTED) DESIGN PRINCIPLES AND BAD SMELLS

QUICK OVERVIEW

MASTER 1 ICE, 2017-2018

BENOIT COMBEMALE
PROFESSOR, UNIV. TOULOUSE, FRANCE

[HTTP://COMBEMALE.FR](http://combemale.fr)
[BENOIT.COMBEMALE@IRIT.FR](mailto:benoit.combemale@irit.fr)
[@BCOMBEMALE](https://twitter.com/BCOMBEMALE)



Objectif du cours

- Connaitre les principes fondamentaux de la conception Objet
- Prendre conscience de l'intérêt de la conception Objet (mais aussi de sa difficulté) par rapport à n'importe quelle technologie de mise en oeuvre
- Savoir faire les bons choix de conception et comprendre leurs intérêts

SOLID: Principles of Class Design

- Single Responsibility Principle (SRP)
- Open/Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
 - ➡ a.k.a. Design by Contract
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

Single-Responsibility Principle (SRP)

- « A class should have one, and only one, reason to change »
- « There should never be more than one reason for a class to change »
- Principe qui semble facile et plein de bon sens
 - Pas si simple dans la vraie vie
 - Compromis à faire avec la complexité, les répétitions et l'opacité

Open-Closed Principle (OCP)

- « Software entities should be open for extension, but closed for modification », *B. Meyer (1988), quoted by R. Martin (1996)*
- « You should be able to extend a class behavior, without modifying it »
- Un code doit être "ouvert à l'extension",
 - l'évolution du logiciel doit se faire de façon incrémentale
- mais "fermé à la modification".
 - sans modifier une ligne de source existante
- Une approche de ce principe se fait par les design patterns "template method" et "strategy"

Heuristiques pour l'OCP

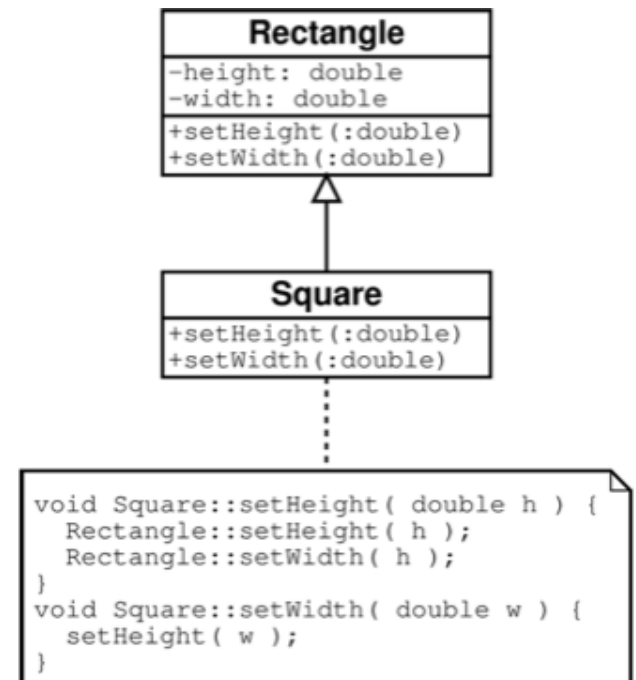
- Mettre toutes les données d'un objet privées
- Pas de variables globales!
- Une modification d'une donnée publique est toujours un risque d'ouvrir un module
 - Il peut y avoir un effet de bord;
 - Les erreurs peuvent être complexes à trouver et fixer
 - Les patches peuvent créer des erreurs ailleurs
- RTTI (Run-Time Type Information) est une mauvaise pratique dangereuse

Liskov Substitution Principle (LSP)

- « All derived classes must be substitutable for their base classes », Barbara Liskov, 1988
- The "Design-by-Contract" formulation: « All derived classes must honour the contracts of their base classes », Bertrand Meyer

LSP: le problème du rectangle carré

- Les clients (users) d'un Rectangle s'attendent à que la mise à jour de la hauteur n'est pas d'impact sur la largeur (et vice versa)
- Le carré ne respecte pas cette attente
- Les programmes clients peuvent être en erreur



LSP: violation de contrat

- Le contrat du rectangle:
 - Hauteur et largeur indépendantes. Il est possible de modifier l'une sans modifier l'autre
- Le carré viole ce contrat
- Les méthodes dérivées ne doivent pas attendre plus et fournir moins que les méthodes de la classe de base
 - Pré conditions ne sont pas plus forte
 - Post conditions ne sont pas plus faibles

LSP: résumé

- Les classes dérivées doivent être pleinement substituables à leurs classes de base
- Guide pour la conception et les choix d'abstraction
- Les bonnes abstractions ne sont pas toujours intuitives
- Violier le principe LSP peut casser le principe d'OCP
 - Besoin de RTTI et utilisation de if/switch
- L'héritage et le polymorphisme sont des outils puissants
 - À utiliser avec attention
- EST-UN est une relation avec une sémantique très particulière en conception objet

Rappel

- Héritage != Sous-typage
- Adaptation != Substitution
- `extends` != `implements`

Dependency Inversion Principle (DIP)

- « Details should depend on abstractions. Abstractions should not depend on details. », *Robert Martin*
- « High level modules should not depend upon low level modules. Both should depend on abstractions », *Robert Martin*
- Why dependency inversion?
 - In OO we have ways to invert the direction of dependencies, i.e. class inheritance and object polymorphism.

Résolution de dépendances

- Ancienne façon
 - Utilisation du 'new' pour créer les dépendances
 - Registre de service
 - Utilisation d'un registre de service pour obtenir ses dépendances
 - Injection de dépendances
 - Les dépendances sont fournies par l'environnement
- ➡ Chacune de ces solutions implique un certain couplage

Ancienne façon (classique)

```
public class Foo {  
  
    private IBar bar;  
    private IBaz baz;  
  
    public Foo() {  
        bar = new SomeBar();  
        baz = new SomeBaz();  
    }  
}
```

- Contre
 - Dépendance entre votre classe et ses classes de dépendance
 - Votre classe doit connaître comment assembler les instances de ses dépendances
 - Très difficile de changer le code sans modifier le code source
 - Très difficile de tester quand vous devez utiliser des stubs ou des mocks.
- Pour
 - Facile à comprendre

Registre de service

```
public class Foo {  
  
    private IBar bar;  
    private IBaz baz;  
    private IServiceLocator locator;  
  
    public Foo(IServiceLocator locator_) {  
        locator = locator_;  
        bar = locator.Get(  
            ServiceNames.BAR  
        );  
        baz = new SomeBaz(  
            ServicesNames.BAZ  
        );  
    }  
}
```

- Contre
 - Votre classe dépend du registre de service
 - Vous devez toujours obtenir le registre de services – soit statiquement ou via ... une sorte de mécanisme d'injection de dépendances
- Pour
 - Facile à comprendre
 - Testable
 - Flexible
 - Extensible
 - Force une meilleure séparation entre interfaces et implémentations

Inversion de contrôle

```
public class Foo {  
    private IBar bar;  
    private IBaz baz;  
    public Foo(Bar bar_, IBaz baz_) {  
        bar = bar_;  
        baz = baz_;  
    }  
}
```

- Contre
 - Vous devez créer les dépendances et les passer pour créer votre schéma d'instance
- Pour
 - Facile à comprendre
 - Testable
 - Flexible
 - Extensible
 - Force une meilleure séparation entre les interfaces et les implémentations
 - Code propre, clair et simple à comprendre

Inversion de contrôle

- *Inversion Of Control, Dependency Injection, The Hollywood Principal, etc.*
- Instead of instantiating concrete class references in your class, depend on an abstraction and allow your concrete dependencies to be given to you.
- *A la place d'instancier une classe concrète (implémentation) dans votre classe, il vaut mieux dépendre d'une abstraction (Interface) et permettre que l'implémentation concrète vous soit fournie*

Types d'injection de dépendances

- Injection par *Setter*
 - Passe les dépendances par les setter/modificateurs de propriétés
- Injection par constructeur
 - Passe les dépendances par les constructeurs

Sans IoC

```
public class WithoutIoC
{
    private IDoSomething somethingDoer;

    public WithoutIoC()
    {
        somethingDoer = new SomethingSpecificDoer();
    }

    public void DoMyThing()
    {
        Console.WriteLine("In the beginnig was me.");
        somethingDoer.Something();
        Console.WriteLine("After my doer did it");
    }
}
```

Dépendances avec les classes concrètes

```
public class WithoutIoC
{
    private IDoSomething somethingDoer;

    public WithoutIoC()
    {
        somethingDoer = new SomethingSpecificDoer();
    }

    public void DoMyThing()
    {
        Console.WriteLine("In the beginnig was me.");
        somethingDoer.Something();
        Console.WriteLine("After my doer did it");
    }
}
```

Injection par constructeur

```
public class WithIoC
{
    private IDoSomething somethingDoer;

    public WithIoC(IDoSomething somethingDoer)
    {
        this.somethingDoer = somethingDoer;
    }

    public void DoMyThing()
    {
        Console.WriteLine("In the beginnig was me.");
        somethingDoer.Something();
        Console.WriteLine("After my doer did it");
    }
}
```

Injection par *setter*

```
public class WithIoCPropertySetter
{
    private IDoSomething somethingDoer;

    public IDoSomething SomethingDoer
    {
        set { somethingDoer = value; }
    }

    public void DoMyThing()
    {
        Console.WriteLine("In the beginnig was me.");
        somethingDoer.Something();
        Console.WriteLine("After my doer did it");
    }
}
```

DIP: résumé

I. High-level modules should *not* depend on low-level modules.

Both should depend on abstractions.

II. Abstractions should not depend on details.

Details should depend on abstractions

R. Martin, 1996

- Une classe de base dans une hiérarchie d'héritage ne doit pas connaître ses sous classes
- *OCP définit l'objectif*
- *DIP définit les mécanismes pour atteindre ses objectifs*
- *LSP offre une garantie pour le DIP*

DIP: résumé

- L'inversion de dépendance permet de rendre un code indépendant de ses conditions d'initialisation, et des API précises ses données d'initialisation
- Utile dans les architectures multi couches
- Exemple:
 - Le framework Spring:
<http://www.theserverside.com/tt/articles/article.tss?l=SpringFramework>
 - le projet Pico (<http://www.picocontainer.org/>)
 - le projet Avalon

Interface Segregation Principle (ISP)

- « Make fine grained interfaces that are client specific », *Robert Martin*
- « Clients should not be forced to depend upon interfaces that they do not use », *Robert Martin*
- De nombreuses interfaces spécifiques aux besoins de la classe cliente sont meilleures qu'une interface aux objectifs généraux
- Les clients ne doivent pas dépendre d'interfaces qu'ils n'utilisent pas

Interface Segregation Principle (ISP)

- Un programme ne doit pas dépendre de méthodes qu'il n'utilise pas
- Principe également lié à la cohésion forte
- Conduit à la multiplication d'interfaces très spécifiques et petites.

Summary

- Single Responsibility Principle (SRP)
 - ➡ Une seule raison de changer
- Open-Closed Principle (OCP)
 - ➡ Prévoir l'extension sans devoir modifier le code existant
- Liskov Substitution Principle (LSP)
 - ➡ Les classes dérivées doivent pleinement se substituer à leur classe de base
- Dependency Inversion Principle (DIP)
 - ➡ Dépendance vers les abstractions, pas vers des classes de mises en oeuvre
- Interface Segregation Principle (ISP)
 - ➡ Éclater les interfaces pour contrôler les dépendances

References

- Website

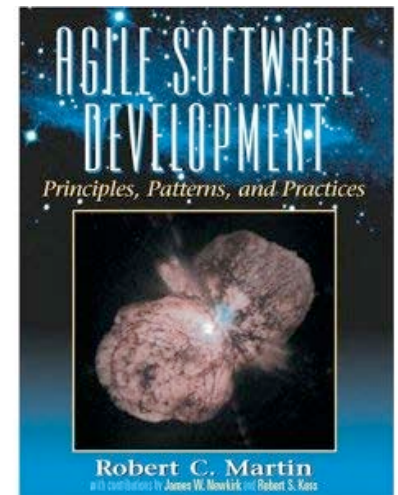
- <http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>

- Robert Martin's '10 Commandments':

- http://groups.google.com/group/comp.object/browse_frm/thread/58808f0dc5c0306f/adee7e5bd99ab111?q=dependency+inversion+group:comp.object&rnum=11&hl=en#adee7e5bd99ab11

- Book:

- « *Agile Software Development, Principles, Patterns, and Practices* », Robert C. Martin, Prentice Hall, 1st ed., 2002.



Principes de conception Objet - Philosophie

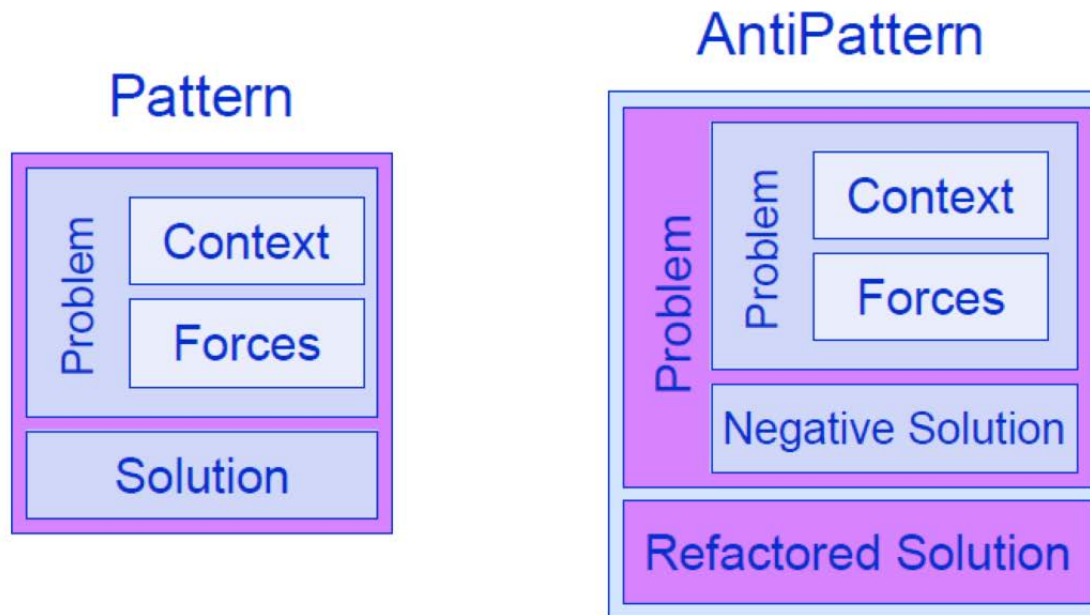
- Il vaut mieux programmer à l'aide d'interfaces plutôt que d'utiliser des classes.
- Les *getters* et les **setters** des objets fournissent un excellent moyen de configurer une application.
- La conception objet est plus importante que n'importe quelle technologie de mise en œuvre
- Un framework ne doit pas forcer les utilisateurs à catcher des exceptions qu'il ne pourra pas récupérer
- La testabilité est essentielle. Un framework d'entreprise doit aider les développeurs à tester leur application.

Conséquences d'une mauvaise conception

- Rigidité
 - Difficulté à changer des parties du code (*continuity*)
 - Réticence aux changements (tout changement devient politique)
- Fragilité
 - Des erreurs surviennent à des endroits inattendus (*protection*)
 - Même de légers changements peuvent causer des erreurs en cascade
- Immobilité
 - Le code est si emmêlé qu'il devient impossible de réutiliser
 - Nombreux codes dupliqués (sémantique ou syntaxique)
- Viscosité
 - Plus facile de faire une grosse verrue que de préserver la conception originale
 - “easy to do the wrong thing, but hard to do the right thing” (R.Martin)

Défauts de conception

- Un **anti-patron** est un type spécial de patron de conception caractérisé par une solution refactorisée



Défauts de conception

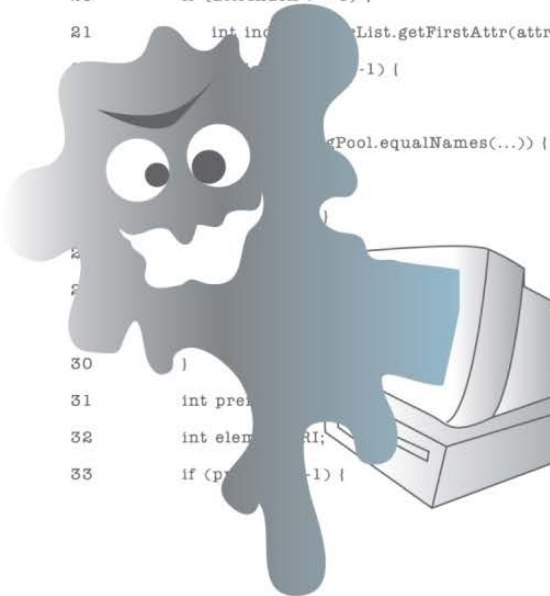
■ 2 exemples d'anti-patrons

■ Blob (God Class)

“ Procedural-style design leads to one object with a lion's share of the responsibilities while most other objects only hold data or execute simple processes ”

- Conception procédurale en programmation OO
- Large classe contrôleur
- Beaucoup d'attributs et méthodes avec une faible cohésion*
- Dépend de classes de données

** À quel point les méthodes sont étroitement liées aux attributs et aux méthodes de la classe.*



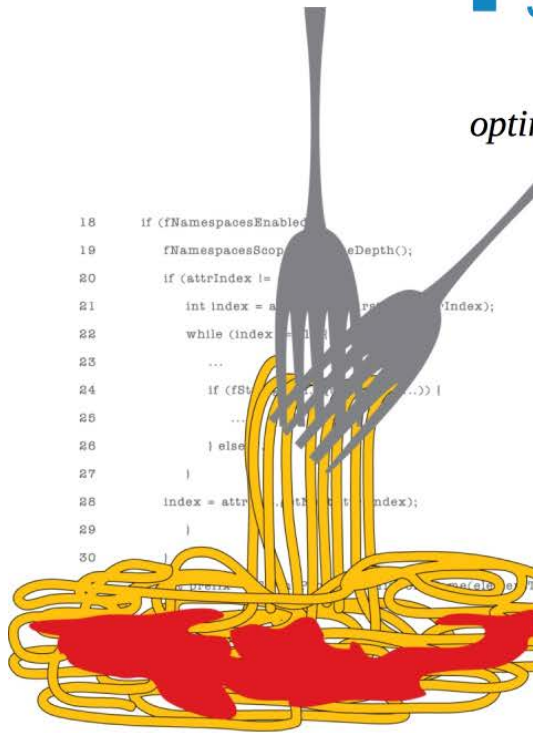
```
18     if (fNamespacesEnabled) {
19         fNamespacesScope.increaseDepth();
20         if (attrIndex != -1) {
21             int index = List.getFirstAttr(attrIndex, -1) {
22                 Pool.equalNames(...) {
23                     ...
24                 }
25             }
26         }
27     }
28 }
29
30 }
31
32 int pre
33 int elem RI;
34
35 if (pre != -1) {
```


Défauts de conception

■ 2 exemples d'anti-patterns

■ Spaghetti Code

“ Ad hoc software structure makes it difficult to extend and optimize code. ”



- Conception procédurale en programmation OO
- Manque de structure : pas d'héritage, pas de réutilisation, pas de polymorphisme
- Noms des classes suggèrent une programmation procédurale
- Longues méthodes sans paramètres avec une faible cohésion
- Utilisation excessive de variables globales

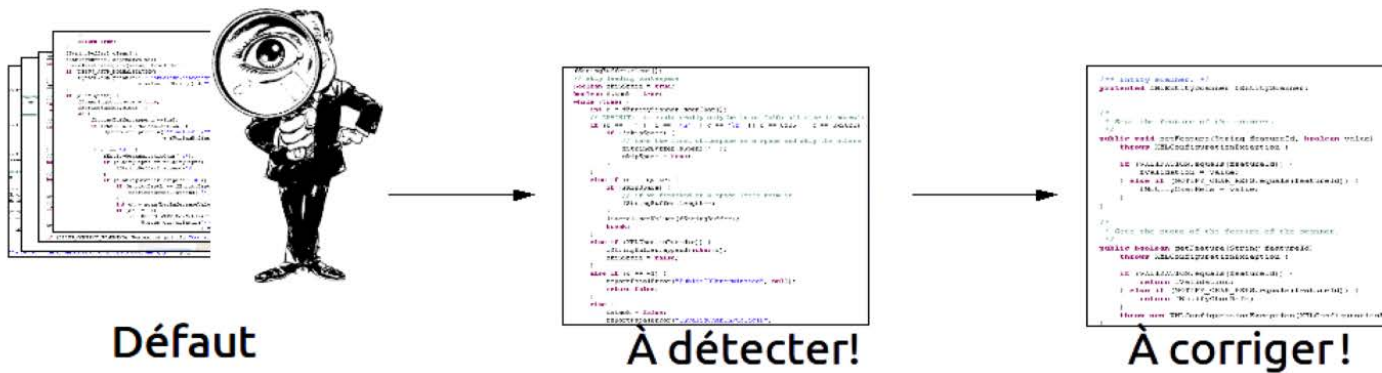
En génie logiciel

un patron \neq de

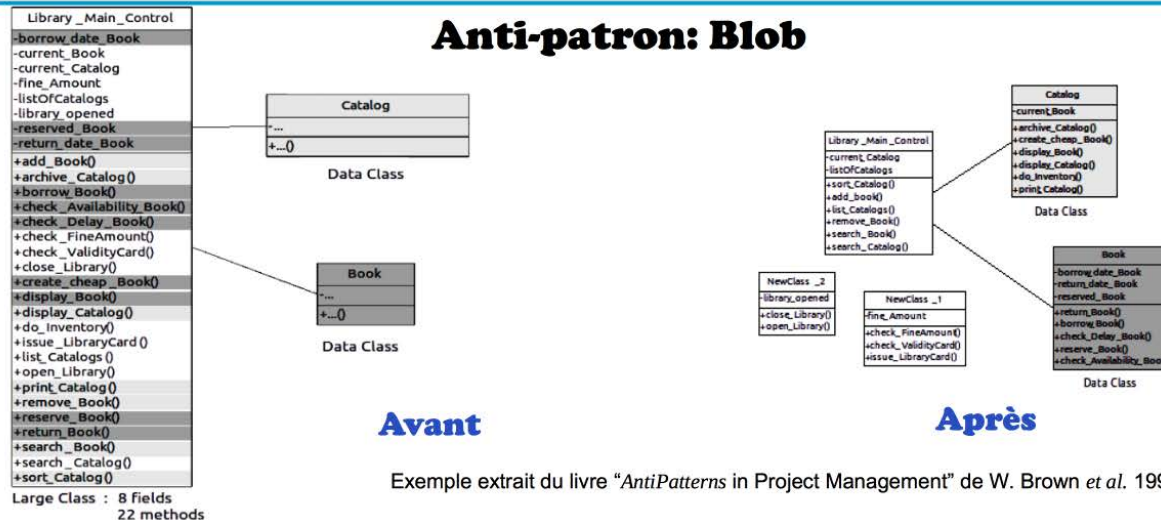


\neq d'un anti-patron

Défauts de conception



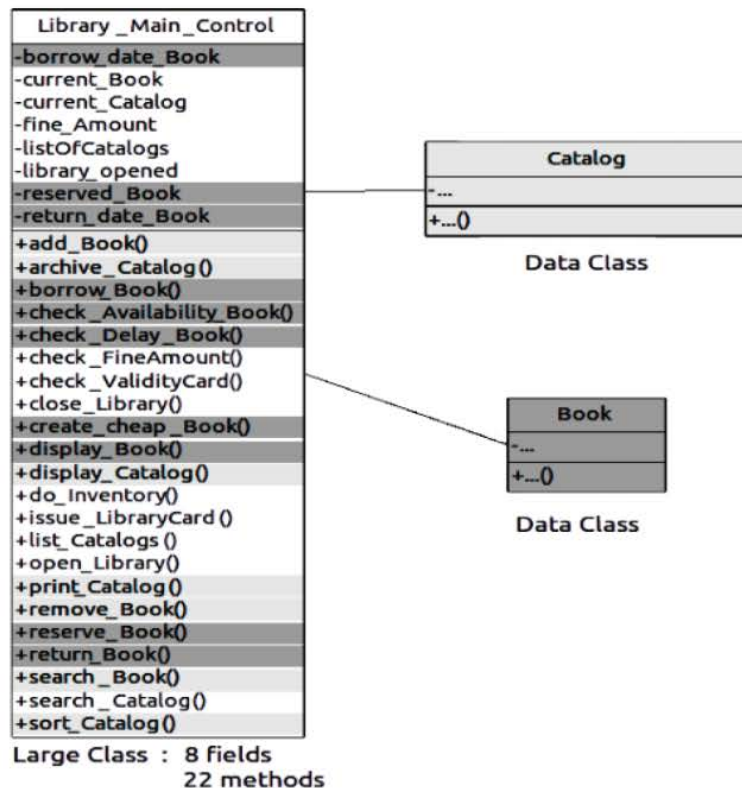
Anti-patron: Blob



Détecter un anti-patron

■ Blob

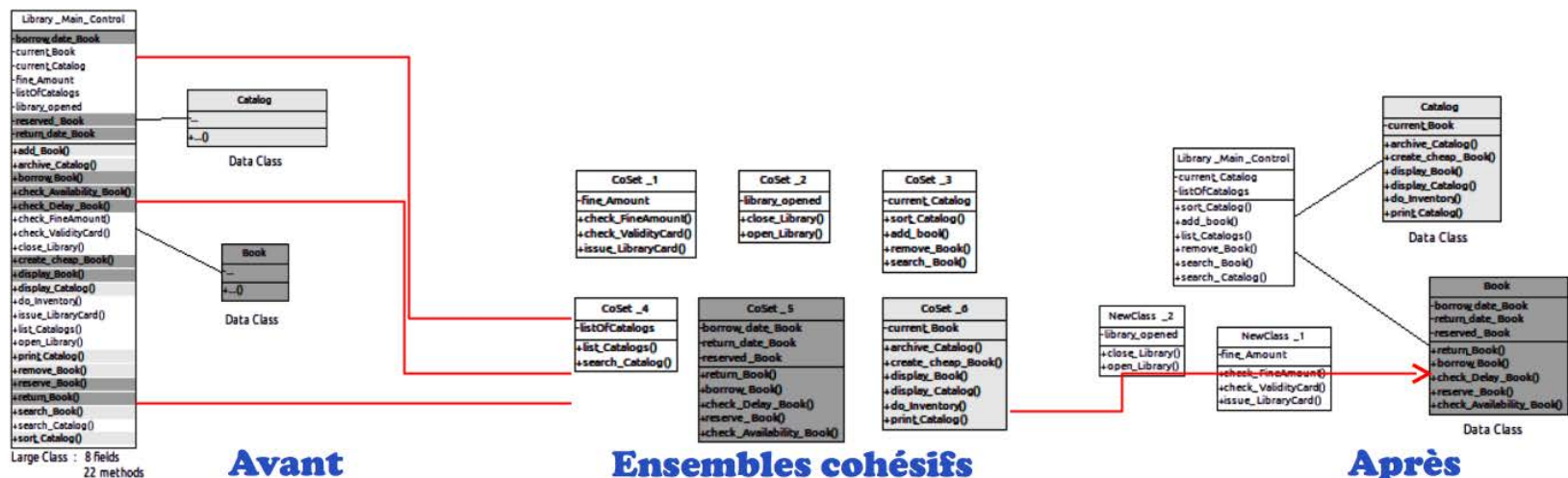
- Identifier les grandes classes
- Identifier les classes de données



Corriger un anti-patron

■ Blob

- Identifier ou catégoriser les attributs et opérations liées
- Rechercher des classes candidates pour les accueillir
- Appliquer des techniques de conception objet (héritage, délégation, patrons, *etc.*)



Mauvaises odeurs

- Longue méthode
- Large classe
- Longue liste de paramètres
- Primitive obsession
- Groupe de données (Data clumps)
- Instructions Switch
- Champ temporaire
- Héritage refusé
- Classes alternatives avec des interfaces différentes
- Hiérarchies parallèles d'héritage
- Classe paresseuse
- Classe de données
- Code dupliqué
- Généralité spéculative
- Chaine de messages
- Middle man
- Feature envy
- Divergent change
- Shotgun surgery
- Classe de librairie incomplète
- Commentaires

■ Divergent Change

- Si une classe est modifiée de différentes manières pour différentes raisons, ça vaut la peine de diviser la classe de sorte que chaque partie soit associée à un type de changement particulier.

■ Shotgun Surgery

- Si un type de changement nécessite plusieurs petits changements de code dans différentes classes, tous ces bouts de code qui sont affectés devraient être mis ensemble dans une classe.

■ Feature Envy

- Une méthode d'une classe est plus intéressée par les attributs d'une autre classe que celles de sa propre classe. Peut-être que placer la méthode dans cette autre classe serait plus appropriée.

```
public class A {  
    public void fooA() {  
    }  
}
```

```
public class B {  
    A a = new A();  
    public void foobarB() {  
    }  
}
```


■ Primitive Obsession

- Parfois, plus intéressant de déplacer un type de données primitives vers une classe légère pour le rendre explicite et identifier les opérations à réaliser (ex : créer une classe date plutôt qu'utiliser un couple d'entiers).

■ Instructions Switch

- Tendent à créer de la duplication. Plusieurs instructions switch éparpillées à différents endroits. Utiliser des classes et du polymorphisme.

■ Hiérarchies parallèles d'héritage

- Deux hiérarchies parallèles existent et un changement dans une classe de la hiérarchie nécessite des changements dans l'autre hiérarchie.

```
public class B extends A {  
}
```

```
public enum AEnum {  
    B,
```

Trouver le défaut

```
class OwnershipTest...
    private void createUserInGroup() {
        GroupManager groupManager = new GroupManager();

        Group group = groupManager.create(TEST_GROUP, false,
            GroupProfile.UNLIMITED_LICENSES, "",
            GroupProfile.ONE_YEAR, null);

        user = userManager.create(USER_NAME, group, USER_NAME,
            "joshua", USER_NAME, LANGUAGE, false, false,
            new Date(), "blah", new Date());
    }
```

Longue liste de paramètres

Trouver le défaut

```
public class Phone {
    public String getAreaCode() {
        return 1 ;
    }
    public String getPrefix() {
        return 21 ;
    }
    public String getNumber() {
        return 1234 ;
    }
}

public class Customer {
    private Phone phone;

    public String getPhoneNumber() {
        return "(" + phone.getAreaCode() + ") "
            + phone.getPrefix() + "-" + phone.getNumber();
    }
}
```

Trouver le défaut

Feature Envy

```
public class Phone {  
    public String getAreaCode() {  
        return 1 ;  
    }  
    public String getPrefix() {  
        return 21 ;  
    }  
    public String getNumber() {  
        return 1234 ;  
    }  
}
```

Customer va rechercher dans
les données de Phone
getPhoneNumber devrait être
La class Phone.

```
public class Customer {  
    private Phone phone;  
  
    public String getPhoneNumber() {  
        return "(" + phone.getAreaCode() + ") "  
            + phone.getPrefix() + "-" + phone.getNumber();  
    }  
}
```

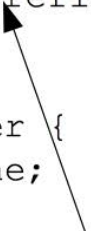
Trouver le défaut

```
public class Phone {  
    public String getAreaCode() {  
        return 1 ;  
    }  
    public String getPrefix() {  
        return 21 ;  
    }  
    public String getNumber() {  
        return 1234 ;  
    }  
}
```

Correction

Customer compte sur Phone
pour faire le formatage

```
    public String toString() {  
        return "(" + phone.getAreaCode() + ") "  
            + phone.getPrefix() + "-" + phone.getNumber();  
    }  
  
public class Customer {  
    private Phone phone;  
  
    public String getPhoneNumber() {  
        return phone ;  
    }  
}
```



```
public abstract class AbstractCollection implements collection
{
    public void addAll(AbstractCollection c) {
        if(c instanceof Set) {
            Set s = (Set)c;
            for(int i=0; i<s.size();i++)
                if(!contains(s.get(i)))
                    add(s.get(i));
        }
        else if(c instanceof List) {
            List l = (List)c;
            for(int i=0;i<l.size();i++)
                if(!contains(l.get(i)))
                    add(l.get(i));
        }
    }
}
```

Trouver le défaut

Instruction Switch

```
public abstract class AbstractCollection implements collection
public void addAll(AbstractCollection c) {
    if(c instanceof Set) {
        Set s = (Set)c;
        for(int i=0; i<s.size();i++)
            if(!contains(s.get(i)))
                add(s.get(i));
    }
    else if(c instanceof List){
        List l = (List)c;
        for(int i=0;i<l.size();i++)
            if(!contains(l.get(i)))
                add(l.get(i));
    }
}
```

**Classes alternatives
avec interfaces différentes**

Code dupliqué

Trouver le défaut

```
public abstract class AbstractCollection {  
    public void add(Object element) {  
    }  
}
```

```
public class Map extends AbstractCollection {  
    // Do nothing because user must input key and value  
    public void add(Object element) {  
    }  
}
```