Bonnes pratiques et Anti-patterns de développement logiciel

Objectifs

- Acquérir les bonnes pratiques de conception objet

- Apprendre les anti-patterns de développement logiciel

Plan

- Maîtriser les principes SOLID
- Appliquer les bonnes pratiques
- Découvrir des anti-patterns de développement

Maîtriser les principes SOLID

SOLID

5 principes de base pour la POO :

- **Single Responsibility (SRP)** : Une classe doit avoir une, et une seule, raison de changer
- **Open / Closed (OCP)** : Vous devriez pouvoir étendre le comportement d'une classe, sans le modifier
- **Liskov Substitution (LSP)** : Les classes dérivées doivent être substituables à leurs classes de base
- **Interface Segregation (ISP) :** Créez des interfaces à granularité fine spécifiques au client."
- **Dependency Injection (DIP)** : Dépend des abstractions, pas des implémentations.

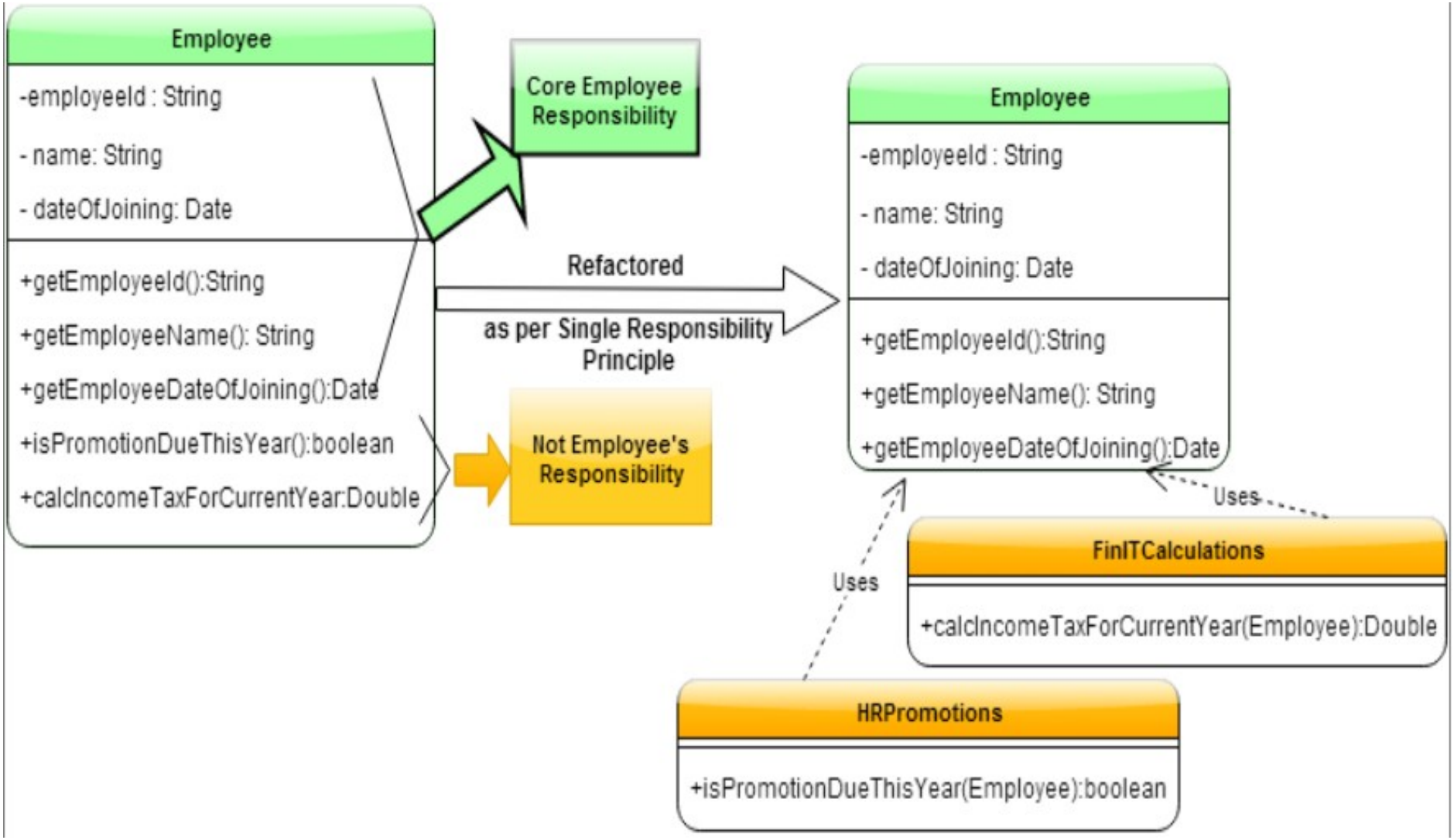
Intérêt : rendre le code plus lisible, facile à maintenir, extensible, réutilisable et sans répétition

SOLID
Single Responsibility Principle

- Le code au sein d'une classe ne doit avoir qu'une seule responsabilité, qu'un seul type de tâche à effectuer.
Si vous prenez conscience que 2 tâches différentes sont effectuées, posez-vous la question de savoir si vous devez scinder votre classe en deux ou non.
(principe applicable aux méthodes, classes, packages, modules)

- **Intérêts :**
- Organisation du code
- Réduire la fragilité (tolérance aux changements)
- Faible couplage
- Faciliter les modifications
- Testabilité + Facilité de debug

SOLID
Single Responsibility



SOLID
Single Responsibility

Comment détecter une violation du SRP ?

- La classe possède beaucoup de dépendances.
- La méthode possède beaucoup de paramètres
- La classe de test devient compliquée
- La classe / méthode est trop longue (>250Lignes)
- Dénomination descriptive
- Classe à faible cohésion
- Le changement à un endroit en brise un autre
- Effet fusil de chasse
- Impossible d'encapsuler un module

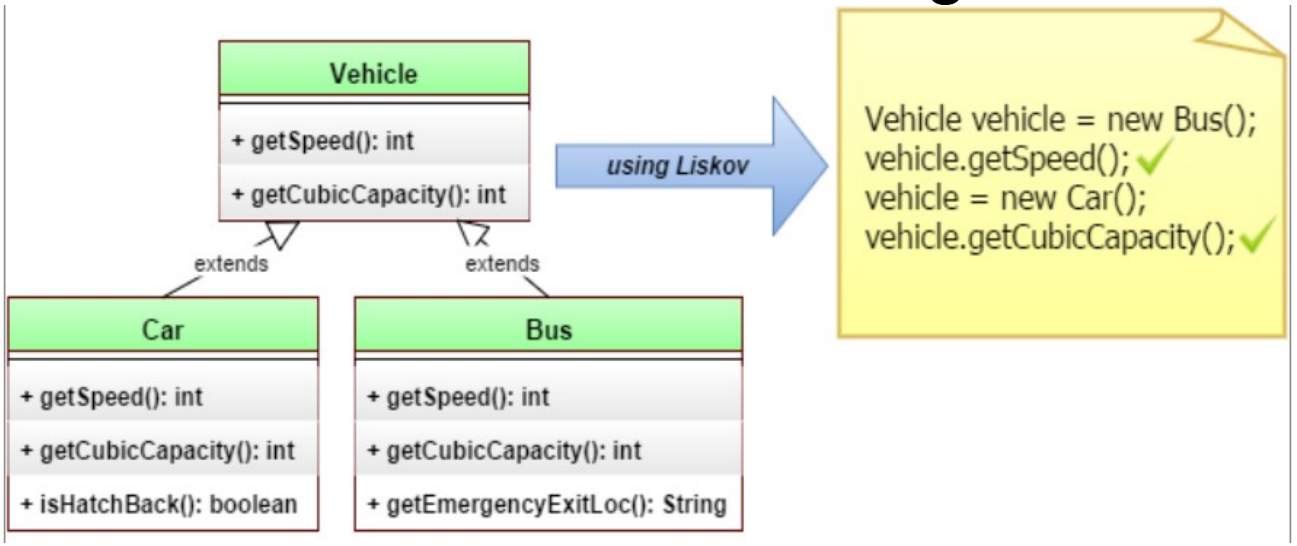
SOLID
Open / Closed

- Une classe doit-être ouverte à une extension/mise à jour, mais doit-être fermée pour toutes modifications.
- **Intérêt** : éviter la régression suite à l'ajout de nouvelles fonctionnalités. Bien sûr qu'il peut y avoir des cas où la modification est indispensable comme la résolution d'un bug, etc. Mais ce principe invite à le faire le moins possible.
- NB. : le design pattern Strategy respecte ce principe.

SOLID
Liskov Substitution

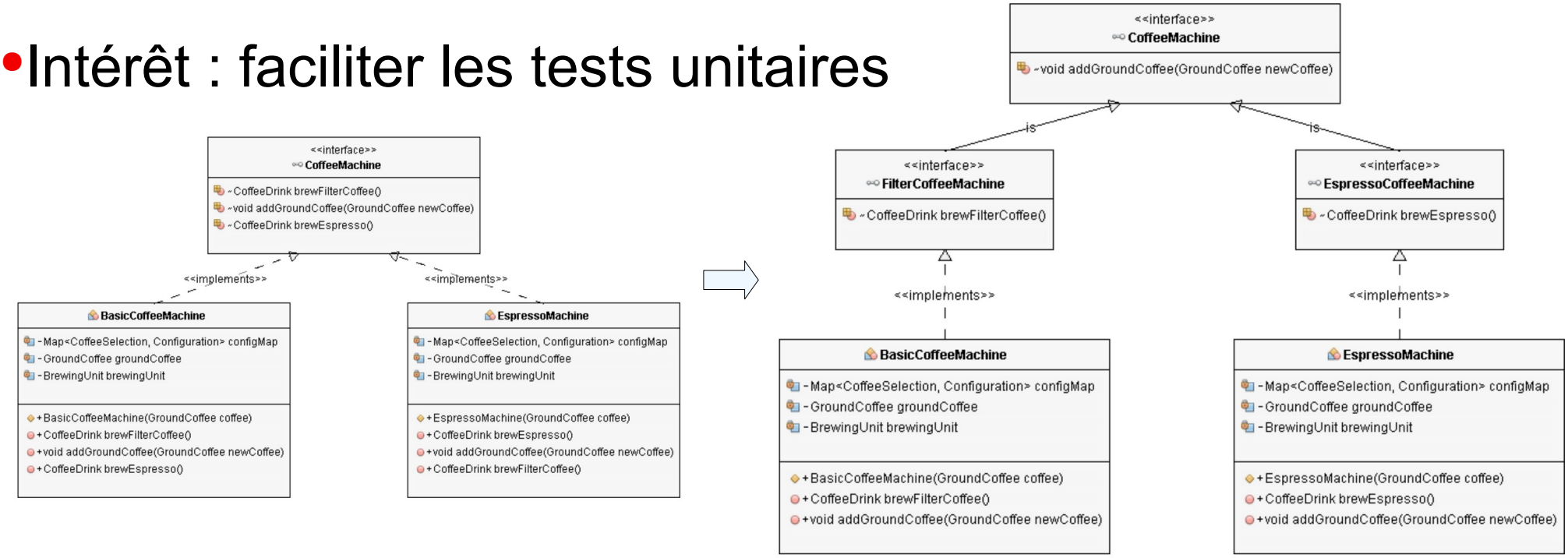
- **Ce principe stipule d'une manière simplifiée, qu'un objet S d'un sous type T doit pouvoir remplacer un objet de type T sans avoir de conséquences dans le comportement du programme.**
(origine Barbara LISKOV – 1987)

- Solution : introduire un niveau d'héritage

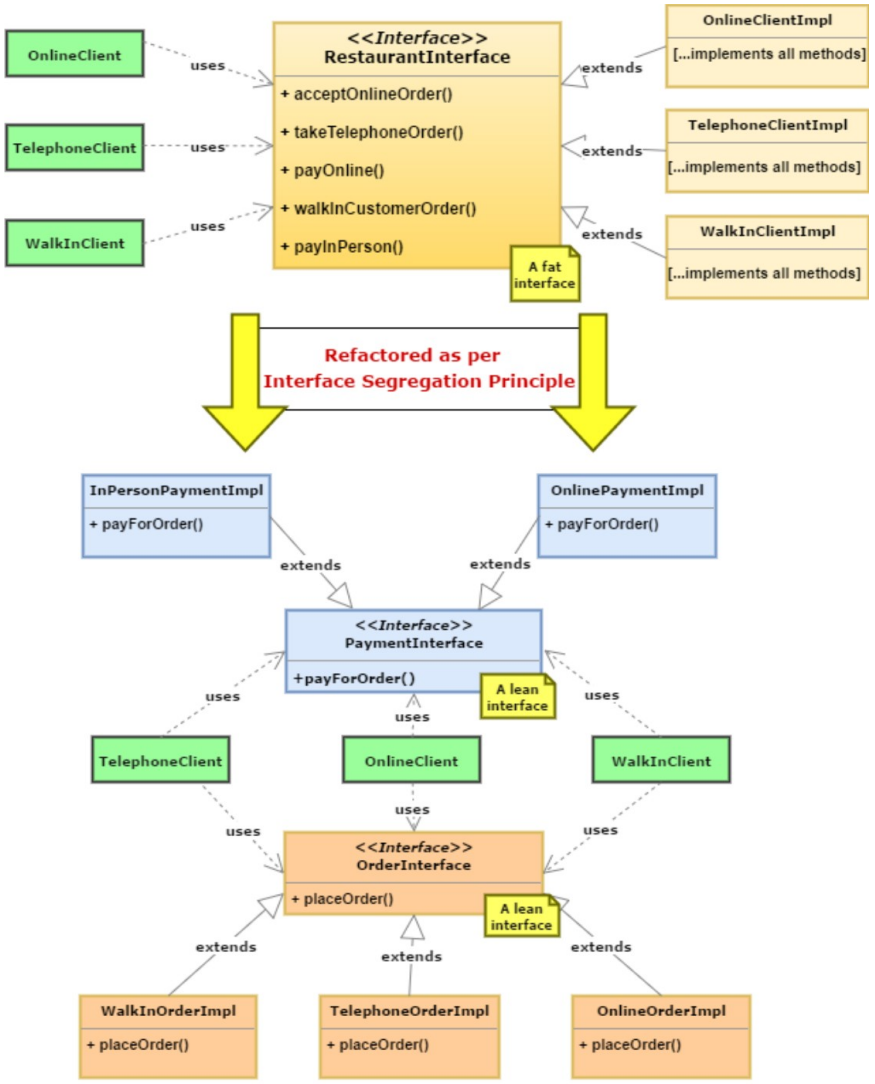


SOLID
Interface segregation

- Technique qui consiste à préférer la création de plusieurs petites interfaces qui concentre vraiment le domaine dans lequel elle a été produite plutôt qu'une seule qui contiendrait beaucoup trop de déclarations.
- Intérêt : faciliter les tests unitaires



SOLID
Interface segregation (2)



SOLID
Dependency Inversion

- Ce principe permet d'inverser les dépendances entre les modules de vos applications.
- Concrètement, si vous avec un projet avec du code métier (Business Layer - BL) et un projet qui s'occupe de la persistance de vos données (Data Access Layer - DAL), généralement votre projet BL référence votre projet DAL.
- Avec ce principe, c'est justement l'inverse qui va se produire, **vous n'allez plus faire référence à DAL dans votre projet BL et vous allez créer une abstraction et c'est à partir de là qu'intervient ce que l'on appelle une injection de dépendance**.

Appliquer les bonnes pratiques

Style de code

- Indenter le code
- Accolades autour des structures de contrôles
- Éviter les opérateurs ternaires
- Éviter les affectations dans les conditions des structures de contrôles
- Documenter le code (Javadoc)

Tell, don’t ask

- Principe permettant de rappeler que l’orienté objet consiste à regrouper des données avec les fonctions qui opèrent sur ces données.

- **Plutôt que de demander à un objet des données et d'agir sur ces données, nous devrions plutôt dire à un objet quoi faire.**

- Dites aux objets ce que vous voulez qu'ils fassent, ne posez pas de questions sur l'état, prenez une décision, puis dites-leur quoi faire :

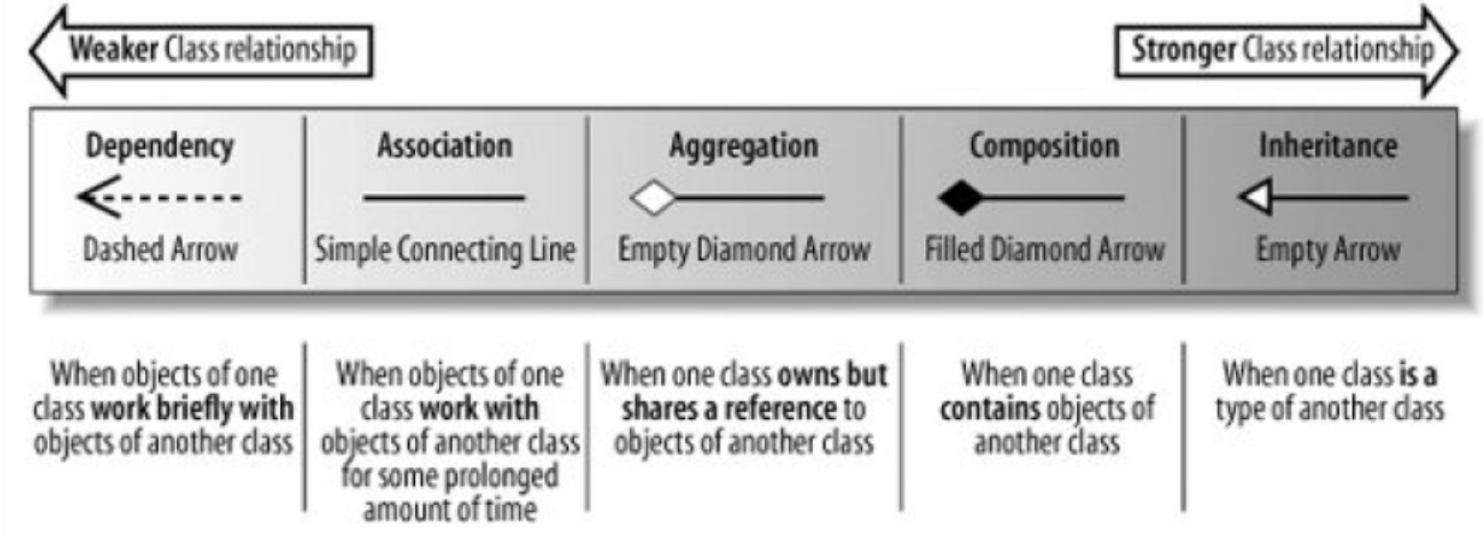
- Pensez de manière déclarative, pas procédurale ;
- Ne demandez pas de carte, puis parcourez la carte ;
- Au lieu de l'itération, appliquez à tous.

Gestion des dépendances

- La POO est centrée sur les objets et les données (au lieu des actions et la logique métier)
- **Une dépendance est une relation entre deux ou plusieurs objets dans laquelle un objet dépend de l'autre ou des autres objets pour sa mise en œuvre. Si l'un de ces objets change, le ou les autres objets peuvent être impactés.**
- Les dégradations (rigidité, fragilité, immobilité) tirent leur origine dans la multiplication des dépendances et de leur architecture : les modules, packages et classes finissent par dépendre les uns des autres aboutissant au code spaghetti.

Dépendances entre types

- **Dépendance** : lorsque les objets d'une classe fonctionnent brièvement avec les objets d'une autre classe.
- **Association** : lorsque les objets d'une classe fonctionnent avec les objets d'une autre classe pendant une période prolongée.
- **Agrégation** : une classe possède une référence aux objets d'une autre classe.
- **Composition** : lorsque une classe contient des objets d'une autre classe.
- **Héritage** : lorsque une classe est un type d'une autre classe.



Composition vs héritage

- Préférer la composition à l'héritage
(polymorphisme à l'exécution => plus de flexibilité)
- L'héritage casse l'encapsulation d'un objet
- La composition permet de remplacer l'implémentation
(si usage d'interface).

Loi de Déméter

- **Chaque unité doit avoir une connaissance limitée des autres unités : elle ne doit voir que les unités étroitement liées à l'unité actuelle.**

- En pratique, cela veut dire qu'une méthode d'une classe peut invoquer les méthodes :
  - de l'objet lui-même
  - des arguments de la méthode
  - des objets créés dans la méthode
  - des propriétés et champs de l'objet

- En suivant la loi de Déméter, votre code sera plus facile à maintenir et donc plus facile à adapter.

Indicateurs d'un code pas terrible (*Code smells*)

- **Rigidité** : le logiciel est difficile à faire évoluer. Une petite modification peut causer une cascade de changements.
- **Fragilité** : le logiciel dysfonctionnement en plusieurs endroits en réponse à un unique changement.
- **Immobilité** : vous ne pouvez pas réutiliser une partie du code dans d'autres projets car cette opération est risquée ou nécessite un grand effort.
- **Complexité inutile**.
- **Répétition inutile**.
- **Opacité** : le code est difficile à comprendre.
-

Métriques

- Longueur de classe : 1000 lignes maxi /classe
- Longueur de méthode : 100 lignes maxi /méthode
- Nombre de paramètres : Une longue liste de paramètres indique que de nouveaux objets devraient être créés pour les encapsuler
- Nombre de champs <= 15/classe
- Nombre de méthodes <= 10/classe
-

Complexité cyclomatique

- On compte certaines instructions.

On commence par 1 à la déclaration
Il faut ensuite ajouter 1 pour chaque if, while, for et case.
Par exemple, la fonction ci-après possède une complexité de 12.

- **Le seuil standard de cette métrique est de 10 points.**

Si vous avez une fonction avec une complexité supérieure à 10, vous devez essayer de la ré-usiner.

Score	Cyclomatic	Risk Type
1 to 10	Simple	Not much risk
11 to 20	Complex	Low risk
21 to 50	Too complex	Medium risk, attention
More than 50	Too complex	Can't test, high risk

```
1  public function example()  {
2    if ($a == $b)  {
3        if ($a1 == $b1) {
             fiddle();
4        } else if ($a2 == $b2) {
             fiddle();
         }  else {
             fiddle();
         }
5    } else if ($c == $d) {
6        while ($c == $d) {
             fiddle();
         }
7     } else if ($e == $f) {
8        for ($n = 0; $n > $h; $n++) {
             fiddle();
         }
     } else{
         switch ($z) {
9            case 1:
                 fiddle();
                 break;
10           case 2:
                 fiddle();
                 break;
11           case 3:
                 fiddle();
                 break;
12           default:
                 fiddle();
                 break;
         }
     }
}
```

Complexité NPATH

- La complexité NPath est le nombre de chemins qui constituent le flux d'une fonction.

```
function foo($a, $b) {
    if ($a > 10) {
        echo 1;
    } else {
        echo 2;
    }
    if ($a > $b) {
        echo 3;
    } else {
        echo 4;
    }
}
```

- Il y a 2 instructions if avec 2 sorties possibles chacune.
On a donc 4 sorties possibles (2 * 2 = 4).

- Cela signifie que la complexité NPath est de 4.
Si nous avions une autre instruction if, nous aurions une complexité de 8, car (2 * 2 * 2 = 8).

- La complexité NPath est exponentielle et peut facilement devenir incontrôlable, dans du vieux code.
Ne soyez pas surpris si vous trouvez des fonctions avec une complexité supérieure à 100 000.

- **La valeur par défaut du seuil de cette métrique est de 200**.
Vous devez rester sous cette valeur.

Complexité cognitive

- Le but n'est pas de mesurer la complexité mathématique de l'algorithme mais de **quantifier l'effort nécessaire pour la comprendre**. Le choix des instructions qui incrémentent suit cette logique en pénalisant les instructions jugées complexes (i.e. goto, continue) et en avantageant les plus lisibles (la définition d'une fonction compte pour 0). La prise en compte de l'indentation suit ce but.

- Par exemple, la complexité cognitive incrémente pour l'instruction switch et chacun des break mais ignore les case alors qu'à l'inverse, la complexité cyclomatique incrémente pour chaque case mais ignore le switch et les breaks.

- L'absence de fondement mathématique, une qualité d'après les auteurs de cette métrique, empêche toute interprétation de la valeur obtenue. La valeur obtenue n'a aucune réalité et ne peut servir qu'à comparer des codes entre eux pour savoir lequel utilise plus de structures ou d'indentation.

- Le seuil proposé est de 15

Métriques de complexité Synthèse

3 métriques permettant de mesurer la complexité du code :
- **La complexité cyclomatique** compte le nombre de points de décision dans le code et donne le nombre minimal de tests pour couvrir le code,
- **La complexité NPath** compte le nombre de chemins acycliques possibles et donne le nombre minimal de tests pour couvrir les combinaisons possibles,
- **La complexité cognitive** calcule un poids qui traduit une difficulté de lecture du code.

- Les utiliser globalement sur un projet n'a pas de sens. Tout d'abord, aucune corrélation avec la qualité du produit n'a été démontrée (*i.e.* nombre de bogues) et on manque effectivement d'études sur cet aspect. Ensuite, il existe des cas où le dépassement des seuils est nécessaire et justifiable. Enfin, la complexité d'un projet est plus liée à sa couverture fonctionnelle qu'au code lui-même (le refactorer ne fera que redistribuer cette complexité entre les différents composants).

- L'utilité de ces méthodes est plutôt de mesurer les portions du code pour les comparer et pointer, parmi la base de code, les endroits les plus pertinents en terme d'amélioration et d'audit. Ces métriques pointent ainsi les zones à risque les plus susceptibles de contenir des défauts voir des vulnérabilités.

- **Il s'agit donc avant tout d'un outil au service de l'auditeur qui, manié avec intelligence, permet d'orienter le travail avec efficience**.

-

Démarche de simplification du code

- le découplage est-il nécessaire pour votre code ? (travailler sur un couplage faible)
- Adopter le minimalisme
- Travailler sur la lisibilité du code (tell don't ask, demeter, documentation…)
- Appliquer éventuellement des design patterns (strategy, factory, …)
- Appliquer les concepts SOLID
- Identifier l'abstraction (ne pas répéter de code)
- Eliminer le code inutile

Codage
Limiter les « return »

- Limiter autant que possible les instructions return par fonction
- Dans l'idéal, une méthode/fonction ne devrait avoir qu'un et un seul point de sortie. Ce devrait être sa dernière instruction.
- L'instruction return fait partie des jump statements. Il y a 3 jump statements en java : continue, break et return. Ces instructions ont une action similaire à celle des instructions goto des langages BASIC ou Fortran : elles transfèrent le contrôle à un autre endroit du programme.
- Pour limiter la complexité, on cherche à minimiser l'usage de ces instructions.

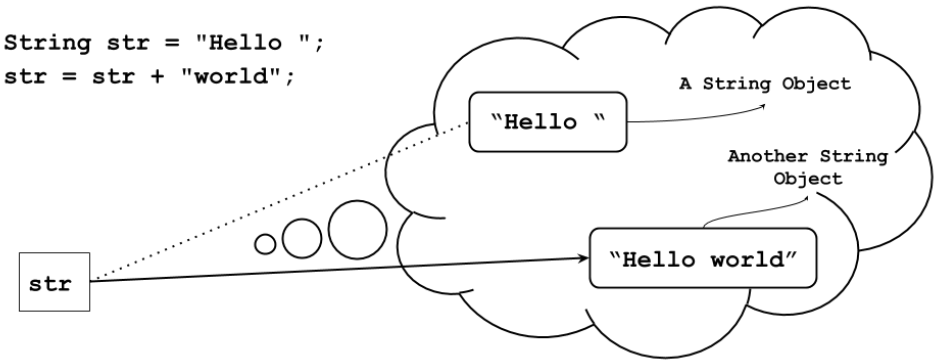
```java
/*
 * Mauvaise pratique (Java).
 */
public class OneReturnOnly1 {
  public String foo(int x) {
    if (x > 0) {
      return "hey";   // oops, multiple exit points!
    }
    return "hi";
  }
}
```

```java
/*
 * Bonne pratique (Java).
 */

public class OneReturnOnly1 {
  String result = "hi";
  public String foo(int x) {
    if (x > 0) {
      result = "hey";
    }
    return result;
  }
}
```

Favoriser StringBuilder à la concaténation

- Concaténation : la chaine est reconstruire à la concaténation



- StringBuilder : non thread safe (plus rapide en monothread)
- StringBuffer : thread safe, synchronized

Les objets StringBuilder et StringBuffer permettent de ne pas reconstruire l'objet mais d'ajouter à la chaine existante :

```
StringBuilder sb = new StringBuilder(‘’Hello”);
sb.append(‘’world ”);
String res = sb.toString();
```

Codage
Eviter les instructions en cascade

- Les instructions en cascade peuvent être sources d'erreurs et compliquent parfois inutilement le débogage des programmes.
- Les piles d'appels et les erreurs de compilations sont bien souvent imprécises lorsqu'on ne respecte pas cette règle.
- Attention toutefois, cette règle est à adapter à votre contexte de programmation car il arrive que ce style soit recommandé dans certains cas précis, **p. ex. pour l'usage des StringBuilder ou des Stream en Java.**

Codage

- **Eviter les conditionnelles négatives**

Les expressions négatives sont plus difficiles à comprendre que les expressions positives.

- **Remplacer les nombres et les chaînes par des constantes**

Il convient d'éviter de conserver les nombres et les chaînes de caractères en brut dans le code.
La bonne pratique est d'utiliser des constantes nommées.

Règles de conception

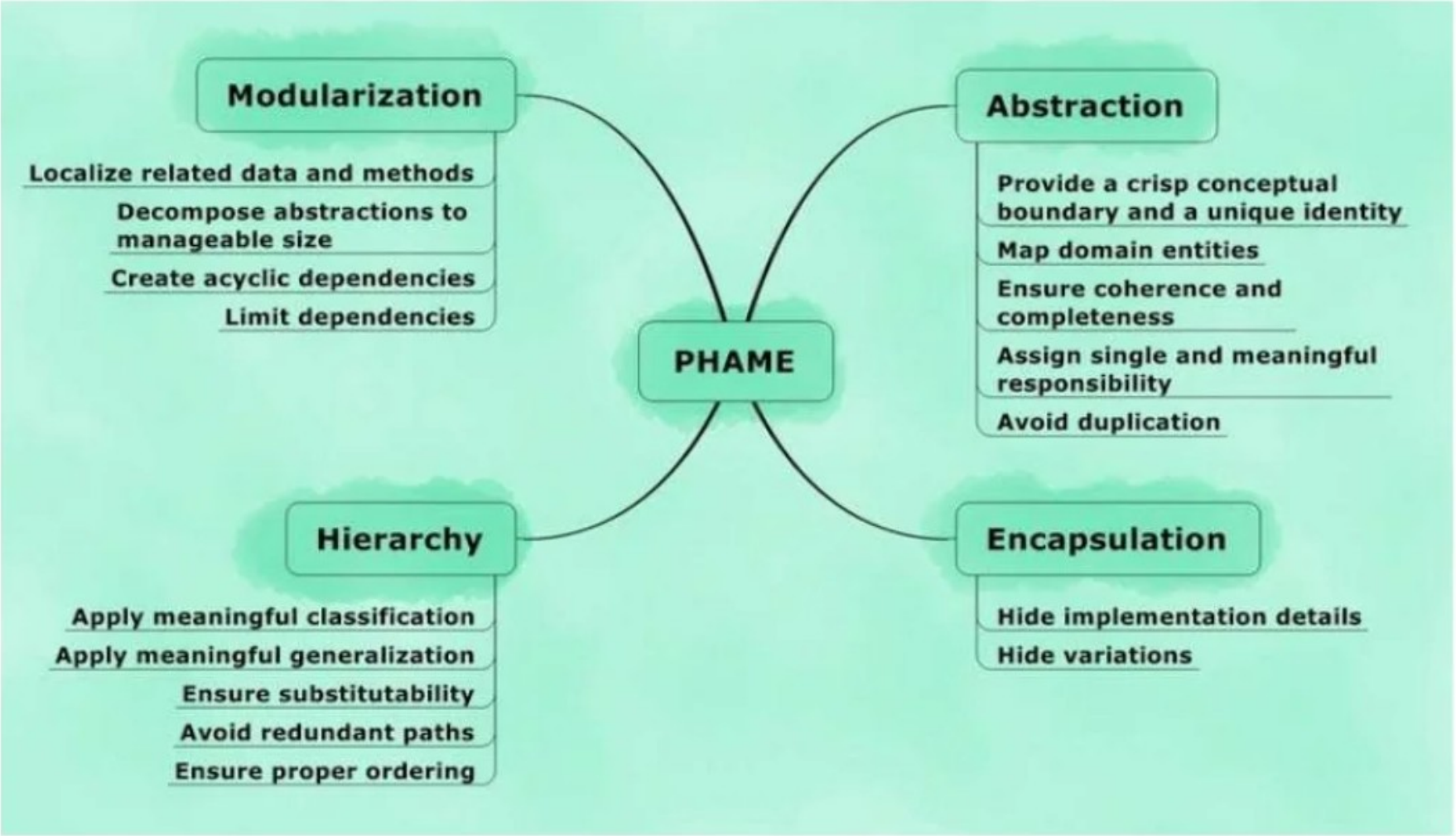
- Si une classe ne contient que des méthodes statiques, il est préférable d'en faire un Singleton
- Substituer les instructions switch lorsqu'elles sont utilisées pour choisir entre plusieurs comportements en fonction d'un type d'objet. Il s'agit de déplacer chaque composante de la condition dans une méthode surchargée au sein d'une sous classe. On rend la méthode originelle abstraite.
- Éviter les champs protégés dans les classes final
- Typer par des interfaces
- Documenter le code

Persistance

- Batch size
- Cache de niveau 2

PHAME

PHAME: Principles of Hierarchy, Abstraction, Modularization and Encapsulation



Anti-patterns

Qu'est-ce qu'un anti-pattern ?

- Un patron est une solution à un problème récurrent et qui facilite les activités périphériques au développement comme la maintenance, l'assurance de la qualité et d'autres.

- Un anti-pattern est une approche de conception ou une construction récurrente qui est inefficace et contreproductive.
=> C'est une mauvaise pratique!

- Mais pourquoi les anti-patterns existent ils?
- Incapacité ou connaissance insuffisante.
- Bonnes solutions à court terme avec des conséquences néfastes à long terme.
- Utilisation de patrons de conception de façon ou dans un contexte inapproprié.

Anti-patterns de code

- DuplicatedCode
- Long Method
- Large Class
- Long ParameterList
- Divergent Change
- ShotgunSurgery
- FeatureEnvy
- Switch Statements
- Message Chains
- InappropriateIntimacy
- RefusedBequest

Anti-patterns de code
Duplicated Code

- **Définition :** Le même code existe à plusieurs endroits dans le système.

- **Refactoring :**
  - Extraire des méthodes
  - Construire une classe générique
  - Extraire des classes
  - Enchaîner les constructeurs
  - Introduire un mécanisme de création d'objet polymorphique grâce au patron FactoryMethod

-

Anti-patterns de code
Long Method

- **Définition :** Une méthode a une longue implémentation qui fait beaucoup de choses.

- **Refactoring :**
  - Extraire des méthodes
  - Décomposer les énoncés conditionnels
  - Assembler une méthode (Compose Method)

-

```
public void add(Object element) {
  if (!readOnly) {
    int newSize = size + 1;
    if (newSize > elements.length) {
      Object[] newElements =
        new Object[elements.length + 10];
      for (int i = 0; i < size; i++)
        newElements[i] = elements[i];
      elements = newElements;
    }
    elements[size++] = element;
  }
}
```

```
public void add(Object element) {
  if (readOnly)
    return;
  if (atCapacity())
    grow();
  addElement(element);
}
```

Anti-patterns de code
Large Class

- **Définition :**
  - Une classe qui a beaucoup de membres ou beaucoup de responsabilités.
- **Refactoring :**
  - Extraire une/des classe(s)
  - Extraire une/des superclasse(s)
  - Extraire une/des sous-classe(s)
-

Anti-patterns de code
Long Parameter List

- **Définition :** Une méthode a une longue liste des paramètres.
- **Refactoring :**
  - Remplacer des paramètres par une méthode.
  - Introduire des objets en paramètres.
-

Anti-patterns de code
Divergent Change

- **Définition :** Une classe change beaucoup, plusieurs fois, ou chaque fois qu'il y a un changement aux exigences.
- **Refactoring :**
  Extraire une/des classes
-

Anti-patterns de code
Shotgun Surgery

- **Définition :** Lorsqu'il y a un changement, on doit changer le code à plusieurs endroits.

- **Refactoring :**
  - Déplacer des méthodes.
  - Imbriquer des classes.

Anti-patterns de code
Feature Envy

- **Définition :** Une méthode utilise plus de membres d'autres classes que de la classe à laquelle elle appartient
- **Refactoring :**
  - Déplacer des méthodes.
  - Extraire et déplacer des attributs ou des méthodes.
-

Anti-patterns de code
Switch statements

- **Définition :** Une méthode est longue et complexe au point de contenir plusieurs instructions «switch» ou conditionnelles.

- **Refactoring :**
  - Remplacer des vérifications de types par des sous-classes.
  - Remplacer des vérifications de types par une instance du patron Strategy ou State.
  - Remplacer des énoncés conditionnels par du polymorphisme.

Anti-patterns de code
Message Chains

- **Définition :** Il existe de longues chaînes d'invocations de méthodes en cascade.
- **Refactoring :**
  - Extraire et déplacer des méthodes.

NB. : non applicable pour les streams et les StringBuilder

-

Anti-patterns de code
Inappropriate Intimacy

- **Définition :** Une classe a beaucoup d'associations avec les membres privés d'une autre classe.

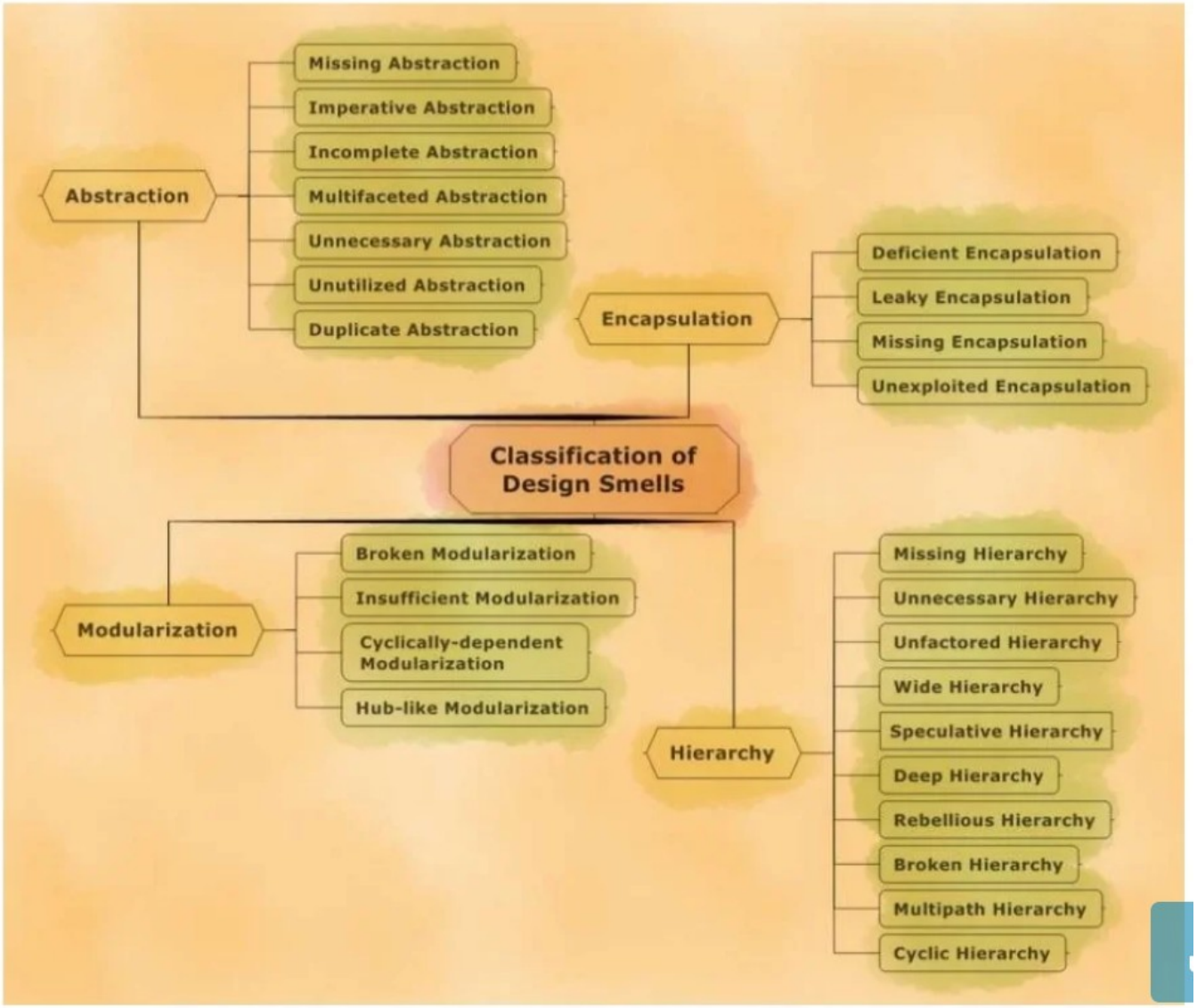
- **Refactoring :**
  - Déplacer des méthodes ou des attributs.
  - Extraire une/des classe(s).
  - Cacher les délégations.
  - Encapsuler un Composite avec un patron Builder.

Anti-patterns de code
Refused Bequest

- **Définition :** Une classe dérivée n'a pas besoin d'hériter de certains membres de la classe de base.

- **Refactoring :**
  - Descendre certaines méthodes ou attributs dans les sous-classes.

Anti-patterns de conception



Anti-patterns de conception

- Missing abstraction
- Multifaceted abstraction
- Duplicate abstraction
- Deficient encapsulation
- Unexploited encapsulation
- Broken modulatization
- Insufficient modularization
- Cyclically-dependend modularization
- Unfactored hierarchy
- Broken hierarchy
- Cyclic hierarchy

Anti-patterns de conception
Missing abstraction

- **Définition :** Lorsque des groupes de données se produisent ensemble souvent. ex. telephoneNumber, areaCode, countryCode

- **Symptômes et conséquences**

- Violation de SRP.
- Complexité augmentée.
- Maintenabilité et compréhensibilité réduites.

- **Refactoring :** Extraire de la classe.
-

Anti-patterns de conception
Multifaceted abstraction

- **Définition :** Une abstraction a plus d'une responsabilité
- **Symptômes et conséquences :**
  - Violation de SRP.
- **Refactoring :** Extraire de la classe ou de la superclasse.
-

Anti-patterns de conception
Duplicate abstraction

- **Définition :** Deux abstractions ont le même nom ou la même implémentation (cloning).
*Le même nom est permis entre deux classes dans des paquets différents.

- **Causes :**
  - Copier-coller
  - Manque de communication

- **Refactoring :**
  Extraire de la classe ou de la superclasse.

-

Anti-patterns de conception
Deficient encapsulation

- **Définition :** Lorsque une abstraction donne plus de permissions qu'il est nécessaire. Ex. lorsque tous les attributs sont déclarés comme publiques.
- **Symptômes et conséquences :**
  - Maintenabilité et sécurité réduites.
- **Causes :**
  - Pensée procédurale dans le contexte orienté-objet.
  - Maintenance négligente.
  - Pour faciliter les tests.
- **Refactoring :** Encapsuler les données et fournir des méthodes d'accès.

Anti-patterns de conception
Unexploited encapsulation

- **Définition :** L'utilisation des vérifications de types explicites lorsque le polymorphisme est disponible.

- **Symptômes et conséquences :**
  - Complexité augmentée.
  - Maintenabilité et compréhensibilité réduites.

- **Causes :**
  - Pensée procédurale dans le contexte orienté-objet.
  - Échec d'application des principes orientés-objet.

- **Refactoring :**
  Remplacer les vérifications de types par du polymorphisme.

Anti-patterns de conception
Broken modularization

- **Définition :** Des données ou des méthodes qui doivent être ensemble, par rapport à la similarité sémantique ou à l'utilisation, font parties de plusieurs abstractions.

- **Symptômes et conséquences :**
  - Haut couplage, faible cohésion
  - Maintenabilité est réduite.
  - Performance se détériore.

- **Causes :**
  - Pensée procédurale dans le contexte orienté-objet.
  - Manque de connaissance de la conception existante.

- **Refactoring :**
  - Déplacer des méthodes ou des attributs
  - Imbriquer des classes

Anti-patterns de conception
Insufficient modularization

- **Définition :** Une abstraction a beaucoup de membres publiques ou des méthodes très complexes.
- **Symptômes et conséquences :**
  - Violation de SRP et d'ISP.
- **Causes :**
  - Contrôle centralisé.
- **Refactoring :**
  Extraire des méthodes ou attributs de la classe ou de l'interface.
-

Anti-patterns de conception
Cyclically-dependend modularization

- **Définition :** Deux abstractions utilisent beaucoup de membres entre elles.
- **Symptômes et conséquences :**
  - Haut couplage
- **Causes :**
  - Passage du pointeur «this»
  - L'abstraction n'est pas conçu correctement.
- **Refactoring :**
  - Déplacer des méthodes ou des attributs
  - Imbriquer des classes

Anti-patterns de conception
Unfactored hierarchy

- **Définition :** Dans une hiérarchie, Il existe de la duplication entre les classes dérivées ou les classes dérivées et la classe de base.

- **Symptômes et conséquences :**
  - Maintenabilité est réduite.

- **Causes :**
  - Duplication de code (Cloning)
  - Copier-coller

- **Refactoring :**
  - Extraire une superclasse
  - Remonter des méthodes ou des attributs

Anti-patterns de conception
Broken hierarchy

- **Définition :** La classe de base et les classes dérivées n'ont pas une relation «est-un».
- **Symptômes et conséquences :**
  - Violation de LSP.
- **Causes :**
  - L'héritage est implémentée pour des raisons d'implémentation, pas de conception.
- **Refactoring :**
  - Remplacer l'héritage par de la délégation
  - Inverser la relation d'héritage
-

Anti-patterns de conception
Cyclic hierarchy

- **Définition :** Une classe de base a une association avec une ou plusieurs de ses classes dérivées.

- **Symptômes et conséquences :**
  - Haut couplage
  - Compréhensibilité réduite.

- **Causes :**
  - L'héritage n'est pas conçu correctement.

- **Refactoring :**
  - Extraire une classe.
  - Déplacer des méthodes.
  - Imbriquer des classes.
  - Implémenter un patron Stratégie ou État.

Plus d'informations sur http://www.dawan.fr

Contactez notre service commercial au
09.72.37.73.73 (prix d'un appel local)