Génie logiciel

Partie 5: Patron de conception/Design Patterns

ELANSARI SOUKAINA
soukaina_elansari@um5.ac.ma

Introduction

- **En conception, il n'est pas rare de faire face à un problème qui a déjà été rencontré et résolu par d'autres personnes.**
- **Réutiliser les solutions trouvées par ces autres personnes permet de gagner non seulement du temps mais aussi de la qualité**

Introduction

- **Pour rendre cette idée concrète, E. Gamma a défini le concept de *patron de conception (design Pattern)*.**
- **Un patron de conception est une solution éprouvée qui permet de résoudre un problème de conception très bien identifié.**
- **Soulignons qu'un patron de conception est un couple <problème/solution>.**

Patron de conception

- **Les Design Patterns permettent d'améliorer la qualité de développement et d'en diminuer la durée.**
- **En effet, leur application réduit les couplages (points de dépendance) au sein d'une application, apporte de la souplesse, favorise la maintenance et d'une manière générale aide à respecter de "bonnes pratiques" de développement.**

Patron de conception

- **E. Gamma a défini plus d'une vingtaine de patrons de conception de référence, qui sont toujours utilisés à l'heure actuelle.**
- **Depuis lors, le nombre de patrons de conception reconnus ne cesse d'augmenter.**
- **De ce fait, il est très important aujourd'hui, lorsque nous faisons face à un problème de conception, de vérifier si un patron de conception n'a pas déjà été défini pour traiter ce problème.**

Couplage fort

- **Couplage fort : Le couplage est une mesure de la dépendance entre classes. Le faible couplage a pour objectif de faciliter la maintenance en minimisant les dépendances entre éléments.**

Couplage fort

- **Le couplage exprime la relation étroite qu'un élément**
- **Un élément faiblement couplé ne possède pas ou peu de dépendances vis a vis d'autres éléments.**

Couplage fort

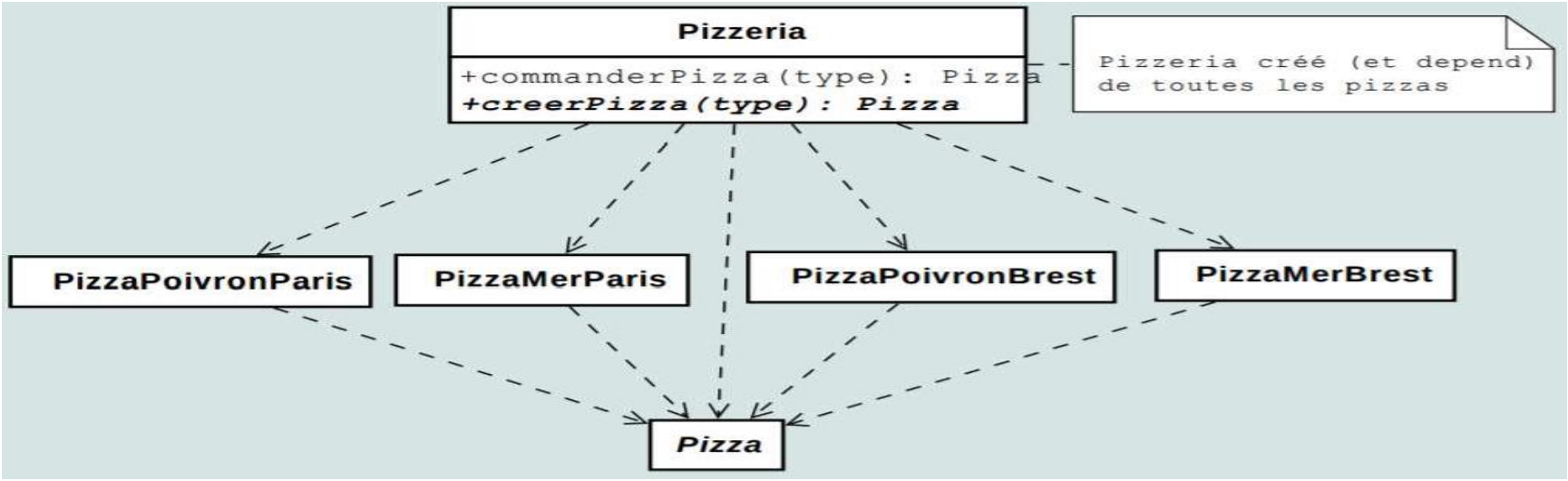
- **Le couplage exprime la relation étroite qu'un élément**
- **Un élément faiblement couplé ne possède pas ou peu de dépendances vis a vis d'autres éléments.**

Couplage fort

- **Un couplage fort génère l'antipattern plat de spaghetti : On ne peut pas déterminer le qui, le quoi et le comment d'une modification de données.**
- **Le composant logiciel est difficilement réutilisable.**
- **Le composant logiciel est difficilement testable.**
- **…**

Patron de conception

Couplage fort



Pizzeria créé les pizzas :

elles dépend donc de toutes les classes décrivant les pizzas,

un changement dans l'une d'elle oblige à modifier Pizzeria.

GoF : *Gang of Four*

Erich Gamma, Richard Helm, Ralph Johnson and John

Classification des patrons de conception

- **GoF ont explicité trois grandes classes de patrons dans leur livre, chacune spécialisée dans :**

- **Création d'objets (creational patterns)**
- **Structure des relations entre objets (structural patterns) Comportement des objets (behavioral patterns)**

Les 23 patterns GoF

Creational	Structural	Behavioral
Abstract Factory	Adapter	Chain of responsibility
Builder	Bridge	Command
Factory Method	Composite	Interpreter
Prototype	Decorator	Iterator
Singleton	Facade	Mediator
	Flyweight	Memento
	Proxy	Observer
		State
		Strategy
		Template Method
		Visitor

Présentation d'un Design Pattern

- **Nom du pattern : utilisé pour décrire le pattern**
- **Problème : description des conditions d'applications. Explication du problème et de son contexte.**
- **Solution : description des éléments (objets, relations, responsabilités, collaboration) permettant de concevoir la solution au problème**
- **Conséquences : description des résultats (effets induits) de l'application du pattern sur le système**

Présentation d'un Design Pattern

- **Un patron décrit une situation fréquente et une réponse éprouvée à cette situation**
- **Le patron peut s'utiliser tel quel ou être adapté ou combiné avec d'autre pour répondre aux besoins.**
- **Les patrons décrits ici sont ceux fournis par le Gof qui sont devenus une référence pour l'ensemble de la communauté des développeurs objet**

Les Design Patterns Créateurs

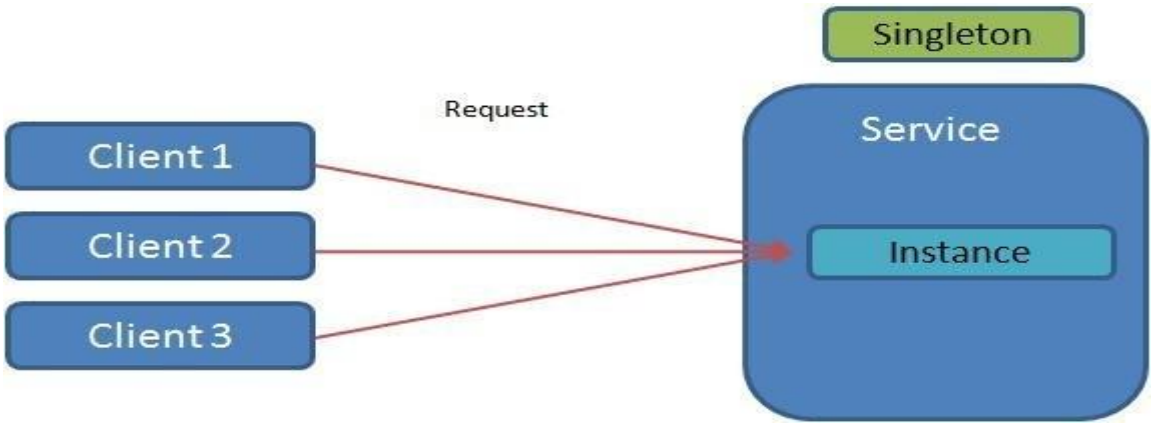
- ☑ Singleton
- ☑ Factory
- ☑ Abstract Factory

Objectif

- **Abstraire le processus d'instanciation.**
- **Encapsuler la connaissance de la classe concrète qui instancie.**
- **Cacher la manière dont les instances sont créées et combines.**

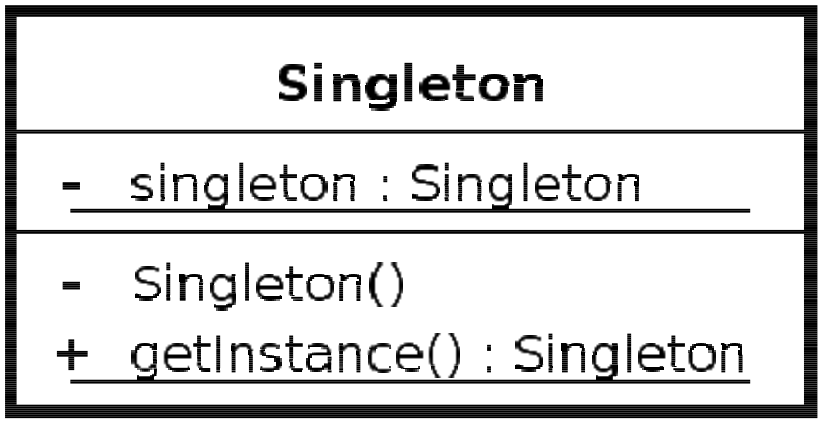
Pattern : Singleton

- **Problème**
  - **Avoir une seule instance d'une classe et pouvoir l'accéder et la manipuler facilement**



Pattern : Singleton

- **Solution**



Pattern : Singleton

- **Le pattern Singleton permet de créer un système qui contrôle l'unicité de la présence d'une instance pour toute la durée de l'application.**
- **La motivation du programmeur est de s'assurer qu'un objet est toujours le même, dans le temps, ou pour mieux gérer la mémoire.**

Pattern : Singleton en Java

```
class Singleton {
   private static Singleton instance = null;

   private Singleton() {
      ...
   }

   public static Singleton getInstance() {
      if (instance == null)
         instance = new Singleton();
      return instance;
   }
   ...
}
```

Pattern : Singleton en Java

```
class Program {
    public void aMethod() {
        Singleton X = Singleton.getInstance();
    }
}
```

Pattern : Singleton en Java

- **Dans un contexte multithread, l'utilisation du pattern Singleton nécessite des précautions pour limiter les problèmes d'accès concurrents.**

- **Il faut synchroniser les appels et les initialisations.**

```
public static synchronized DogService getInstance() {
    if (instance == null) {
        instance = new DogService();
    }
    return instance;
}
```

Pattern : Factory

- **Description du problème**
  - **Il est fréquent de devoir concevoir une classe qui va instancier**
    - **Par exemple une usine va fabriquer des produits en fonction du modèle qu'on lui indique.**

Pattern : Factory

Le problème avec cette implémentation, c'est que la classe correspondant à l'usine va être fortement couplée à tous les produits qu'elle peut instancier car elle fait appel à leur type concret

Pattern : Factory

- **Début de solution**
  - **La première solution est de regrouper l'instanciation de tous les produits dans une seule classe chargée uniquement de ce rôle. On évite alors la duplication de code et on facilite l'évolution au niveau de la gamme des produits.**

Cette solution appelée Fabrique Simple

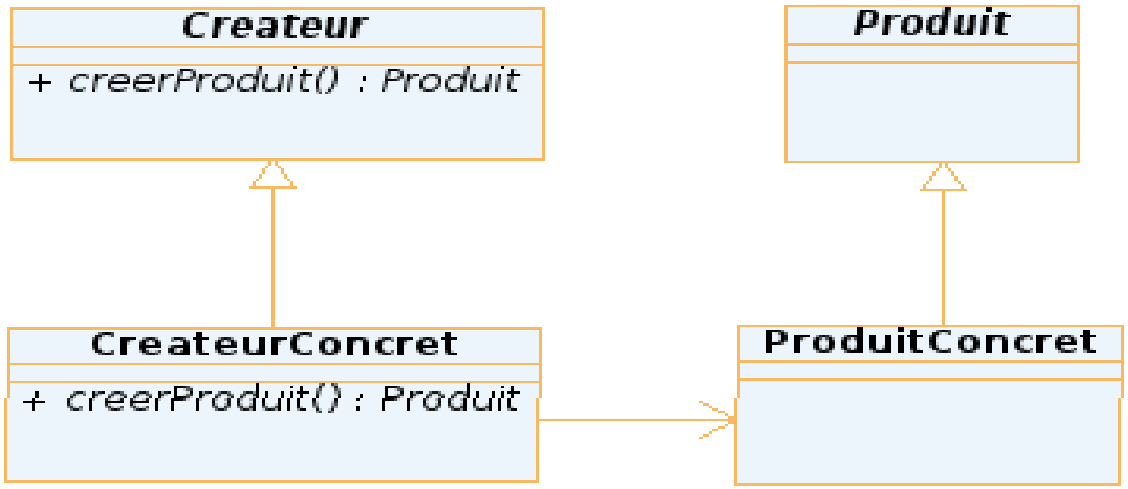
Pattern : Factory

- **Fabrique Simple est une bonne pratique de conception mais pas un design pattern.**
- **Si par la suite l'entreprise évolue et a besoin de plusieurs usines, chacune spécialisée dans la fabrication de certains produits, Fabrique Simple ne va plus suffire.**
- **Dans ce cas il faut prévoir l'utilisation du design pattern Fabrique.**

Pattern : Factory

- **Ce design pattern consiste à donner à une classe particulière, *la fabrique*, la responsabilité de fabriquer les objets d'une autre classe, à savoir *les produits*.**

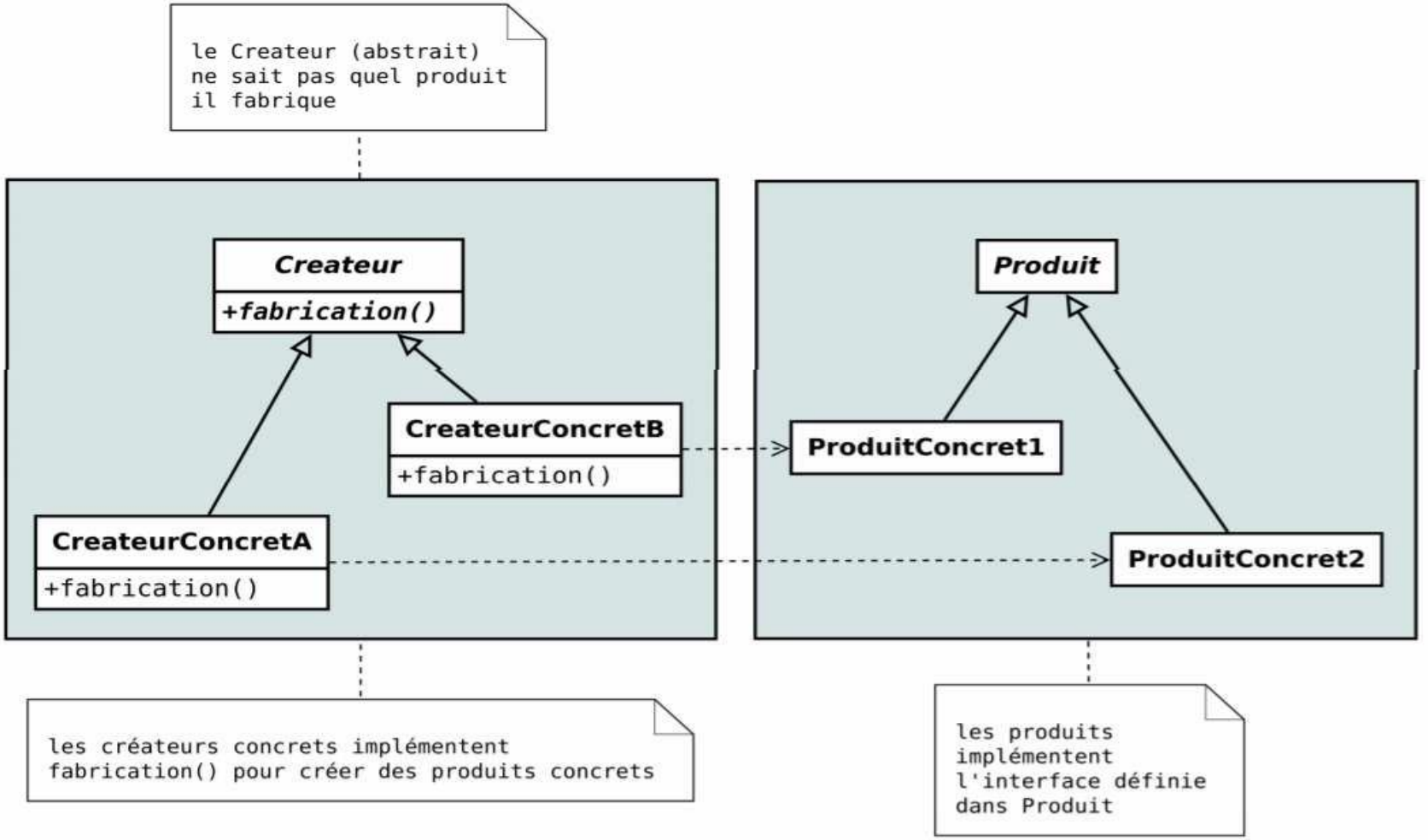
Pattern : Factory



Le créateur contient toutes les méthodes permettant de manipuler les produits exceptée la méthode *creerProduit* qui est abstraite.

Les créateurs concrets implémentent la méthode *creerProduit* qui instancie et retourne les produits. Chaque créateur concret peut donc créer des produits dont il a la responsabilité.

Pattern : Factory



Pattern : Abstract Factory

- **Problème :**
  - **Considérons une boite à outils d'interfaces utilisateur. Pour que l'application soit portable à travers les standards, il ne faut pas que les interfaces soient codées « en dur » pour un standard particulier.**

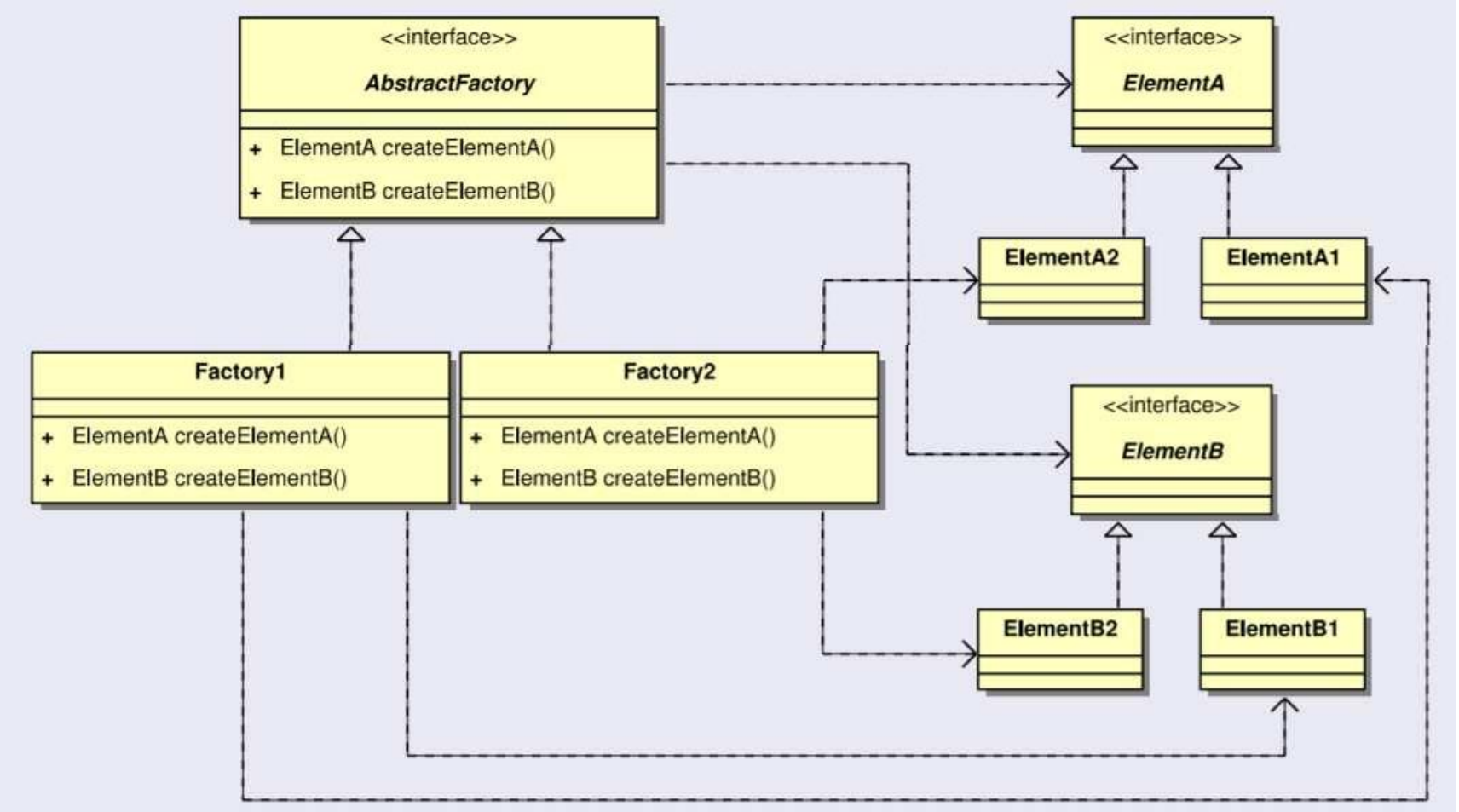
Pattern : Abstract Factory

- **Exemple :**
  - **J'implémente une application de gestion d'assemblage de voitures destinées aux marchés européen et américain. Dans les 2 cas, j'effectue les mêmes tâches comme, par exemple, monter le moteur et les roues. Cependant, j'ai des spécificités pour chacun des marché : par exemple pour le marché français j'utilise un moteur de 110ch et pour le marché américain un moteur de 200ch. Dans ce cas de figure, je souhaiterais simplifier l'implémentation car j'effectue des tâches communes sur des produits de mêmes familles (des voitures) et qui présentent des caractéristiques différentes (donc des implémentations différentes).**

Pattern : Abstract Factory

- **Le but du pattern Abstract Factory est la création d'objets regroupés en familles sans devoir connaître les classes concrètes destinées à la création de ces objets.**

Pattern : Abstract Factory



Résumé : DP de création

- **Le Factory Pattern est utilisé pour choisir et retourner une instance d'une classe parmi un nombre de classes similaires selon une donnée fournie à la factory .**
- **L'Abstract Factory Pattern est utilisé pour retourner un groupe de classes.**
- **Le Singleton Pattern est un pattern qui assure qu'il n'y a qu'une et une seule instance d'un objet et qu'il est possible d'avoir un accès global à cette instance.**

Les Design Patterns de comportement

- **Observateur**

Observateur (Observer)

- **Le pattern Observateur (en anglais *Observer*) définit une relation entre objets de type un-à-plusieurs, de façon que, si un objet change d'état, tous ceux qui en dépendent en soient informés et mis à jour automatiquement.**

Observateur (Observer)

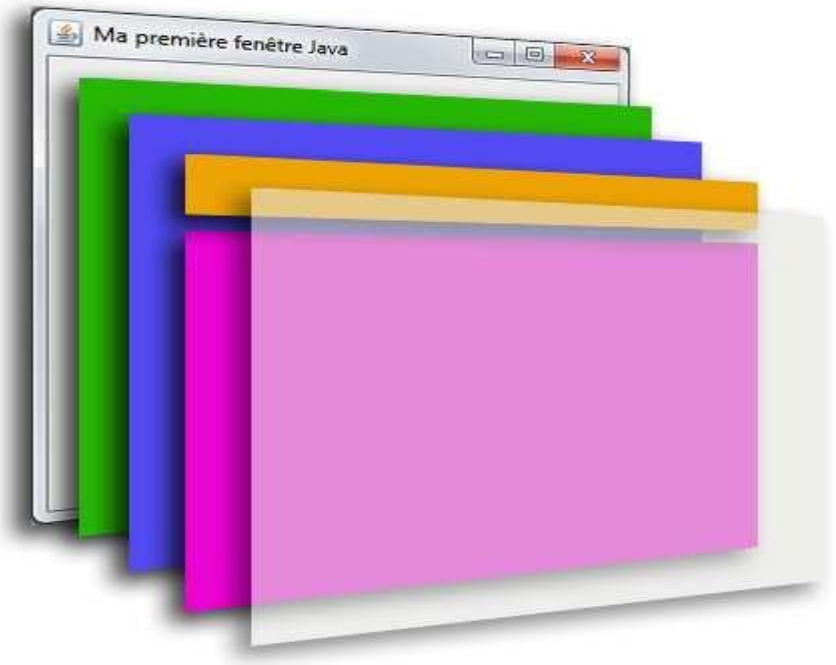
- **Exemple : On considère une classe *HeurePerso* possédant dans le même attribut l'heure et la minute courante.**
  - **Cette classe est utilisé pour l'affichage de l'heure courante dans une fenêtre.**
  - **Pour cela, on définit une classe *AfficheHeure* qui se charge d'afficher l'heure et la minute courante dans une partie de la fenêtre.**
  - **On peut alors se demander quelle démarche adopter pour que la classe chargée de l'affichage (*AfficheHeure*) soit tenue informée en temps réel de l'heure courante stockée dans la classe *HeurePerso* ?**

Observateur (Observer)

- **On peut identifier deux solutions.**
  - **Soit la classe d'affichage se charge de demander à la classe HeurePerso la valeur de son attribut**
  - **soit la classe HeurePerso qui informe la classe AfficheHeure lors de changements.**
- **Il est facile de s'apercevoir que la première solution n'est pas la meilleure.**

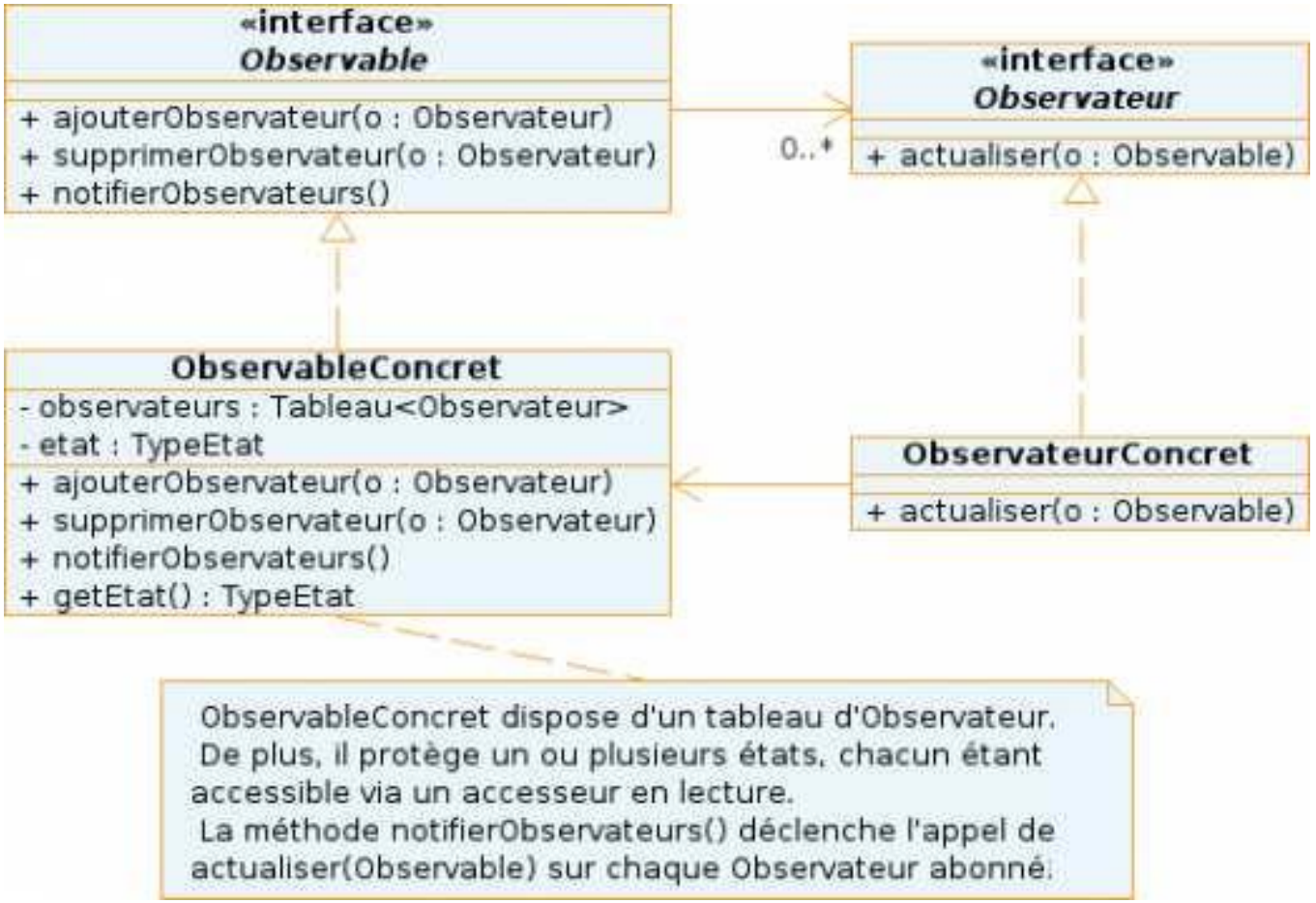
Observateur (Observer)

- **La solution consiste donc à laisser la charge à la classe HeurePerso d'informer sa classe d'affichage de ses changements de valeurs.**



Observateur (Observer)

Diagramme UML



Observateur (Observer)

- **Le diagramme UML du pattern Observateur définit deux interfaces et deux classes.**
  - **L'interface Observateur sera implémenté par toutes classes qui souhaitent avoir le rôle d'observateur.**
  - **une interface Observable qui devra être implémentée par les classes désireuses de posséder des observateurs.**

Observateur (Observer) :Mise en pratique

- **La classe de l'objet « observe » doit étendre la classe « Observable » et ainsi l'objet ajoute un observateur via la méthode addObserver.**
- **Les objets observateurs doivent implémenter l'interface Observer et par conséquent implémenter la méthode *actualiser()*.**

Observateur (Observer) :Mise en pratique

- **Une entreprise (siège) peut avoir ou être liée à plusieurs établissements. On souhaite notifier tout changement survenu dans « Entreprise » aux objets liés, de type « Etablissement » et ajoutés comme observateurs.**

Observateur (Observer) :Mise en pratique

```
public class Entreprise extends Observable {
    private int id;
    private boolean etat;

    public void setEtat(boolean etat) {
        this.etat = etat;
        setChanged();
        notifyObservers(this.etat);
    }
    public boolean isEtat() {
        return etat;
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String toString() {
        return "Entreprise (id=" + id + " etat=" + etat + ")";
    }
}
```

Observateur (Observer) :Mise en pratique

```
public class Etablissement implements Observer {

    private int id;
    private String infos;

    public void update(Observable obs, Object obj) {
        if (obs instanceof Entreprise) {
            setInfos("Notification recu  ");
        }
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getInfos() {
        return infos;
    }
    public void setInfos(String infos) {
        this.infos = infos;
    }
    public String toString() {
        return "Etablissement ( id=" + id + " infos :" + infos + ")";
    }
}
```

Observateur (Observer) :Mise en pratique

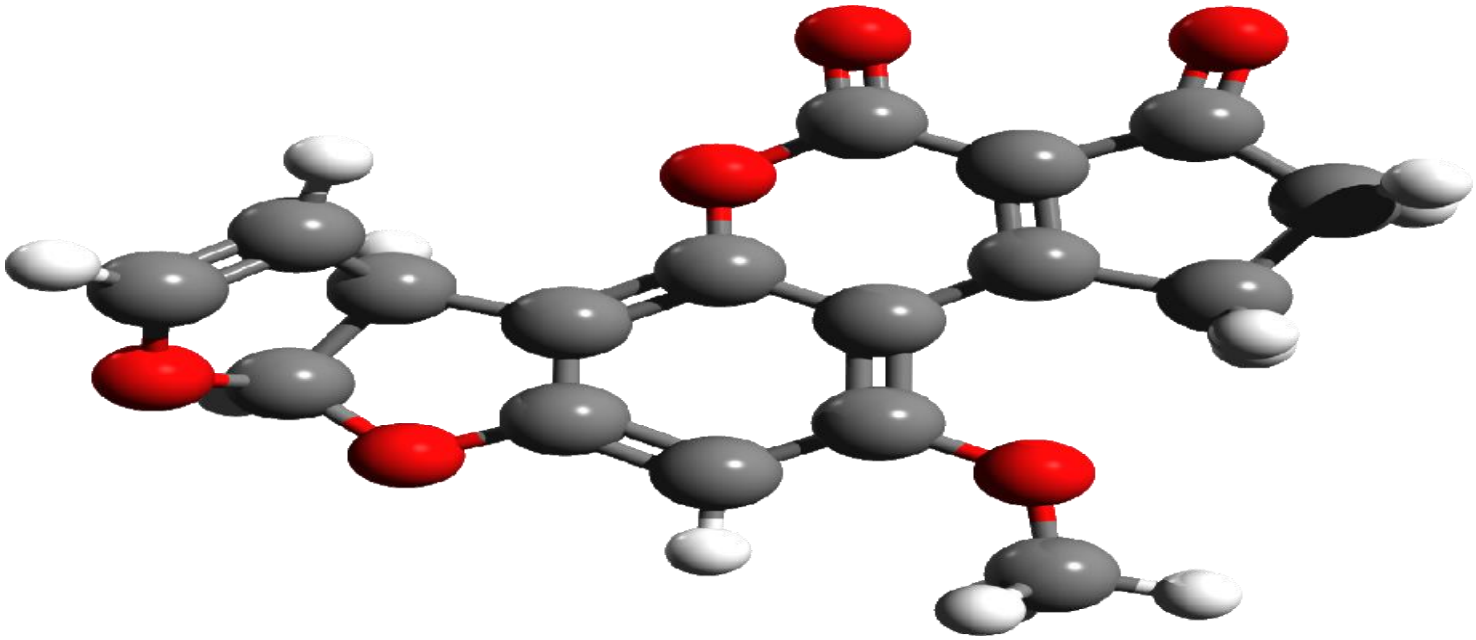
```
public static void main(String[] args) {

    // définir un etablissement
    Etablissement etab = new Etablissement();
    etab.setId(1);
    etab.setInfos("RAS");

    // definir une entreprise
    Entreprise ent = new Entreprise();
    // Ajouter un observer
    ent.addObserver(etab);
    System.out
            .print("Avant mise à jour de l'etat de l'entreprise: \t Etab: "
                    + etab);
    // Effectuer un changement sur l'attribut etat de l'entreprise
    ent.setEtat(true);
    System.out
            .print("Après mise à jour de l'etat de l'entreprise: \t Etab: "
                    + etab);
}
```

Les Design Patterns de Structure

- ☑ **Décorateur**

Les Design Patterns de Structure

- **Abstraction de la manière dont les classes et les objets sont composés pour former des structures plus importantes.**



Design pattern Décorateur

Contexte

- Ajout dynamique de fonctionnalité à un objet

Solution naïve

- Intégrer toutes les fonctionnalités dans la classe
- Utiliser des conditionnelles pour utiliser une fonctionnalité

Augmentation de la complexité du programme

Design pattern Décorateur

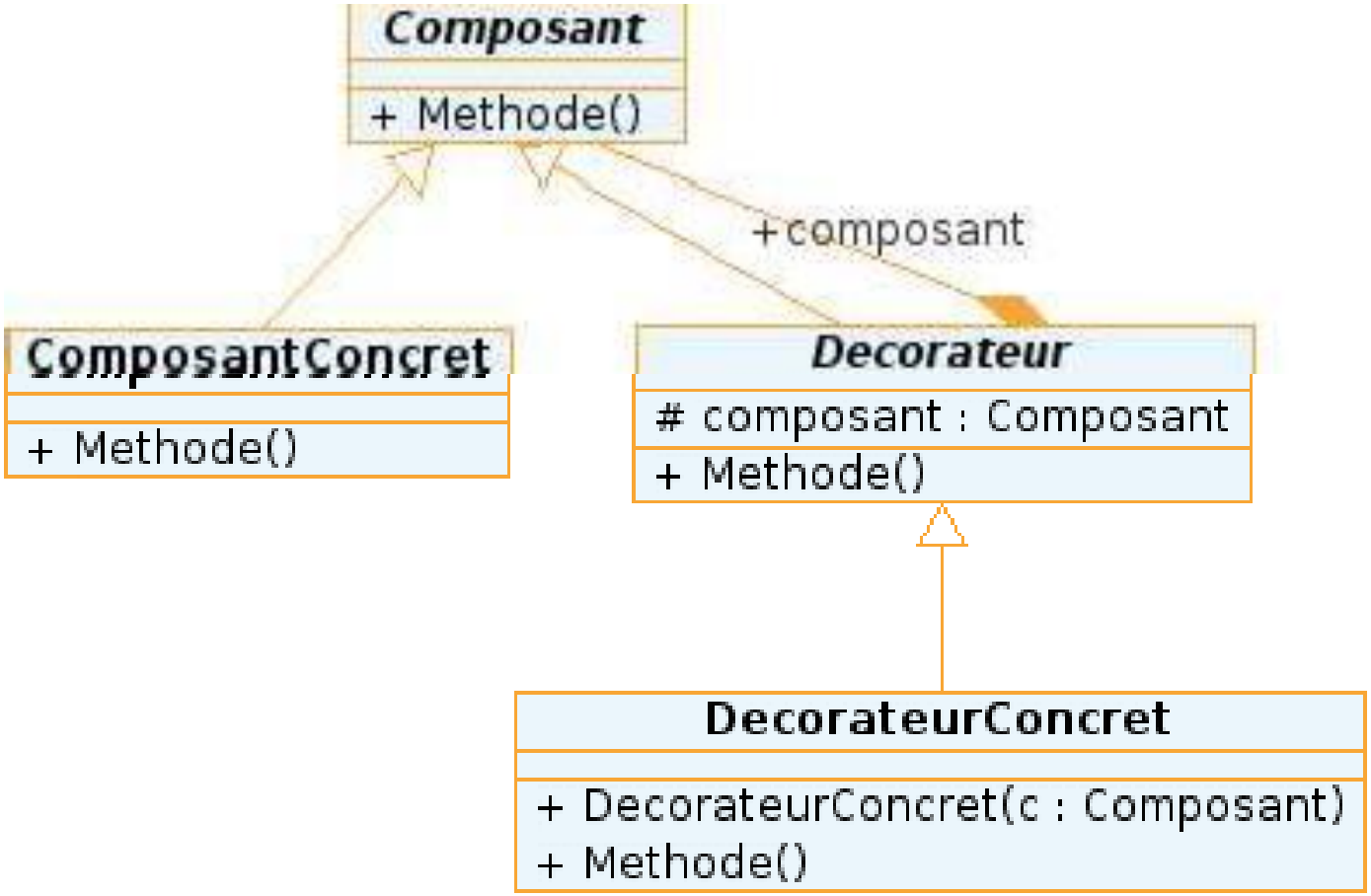
- **Le pattern Decorator est un design pattern de structure qui fait partie des design pattern du GoF (Gang of Four).**

- **Decorator nous permet d'ajouter des fonctionnalités nouvelles à une classe de façon dynamique sans impacter les classes qui l'utilisent ou en héritent.**

Design pattern Décorateur

- **Pattern**
  - **Une interface et une classe quila réalise avec le comportement de base.**
  - **Des classes Decorator qui sont composées d'une instance de l'interface et qui rajoute une fonctionnalité.**

Design pattern Décorateur



Design pattern Décorateur

- **La classe abstraite Composant définit le point de départ de ce diagramme.**
- **Plusieurs ComposantConcret peuvent hériter de Composant.**
- **Si l'on souhaite étendre (ou modifier) l'ensemble des fonctionnalités des ComposantConcret on peut créer un décorateur. Il s'agit d'une classe abstraite héritant de Composant et ayant un attribut de type .**

Design pattern Décorateur

- **Pour ajouter des fonctionnalités à un ensemble de *ComposantConcret* on va créer des classes *DecorateurConcret* qui héritent de *Decorateur***
- **Un *DecorateurConcret* contient un constructeur permettant d'initialiser l'attribut composant présent dans le décorateur.**
- **Il faut ensuite que la classe *DecorateurConcret* redéfinisse la méthode déclarée abstraite dans le décorateur.**

En bref, les Design Patterns

- **C’est...**
  - **une description d’une solution classique à un problème recurrent.**
  - **une description d’une partie de la solution.**
- **Ce n’est pas**
  - **une méthode : Ne guide pas une prise de décision : un pattern est la décision prise.**

Les anti-patrons

- **Les anti-patrons ou *antipatterns* sont des erreurs courantes de conception des logiciels. Leur nom vient du fait que ces erreurs sont apparues dès les phases de conception du logiciel, notamment par l'absence ou la mauvaise utilisation de patrons de conception,**
- **Sont de "mauvaises" solutions, mauvaises pratiques de conception qui mènent à des conséquences négatives.**

Les anti-patrons

- **Les anti-patrons se caractérisent souvent des coûts de réalisation ou de maintenance élevés, des comportements anormaux et la présence de bugs.**
- **Sont moins formalises que les patrons de conception ;**
- **Sont au nombre d'une quarantaine.**

Les anti-patrons

- **Blob, god object**

✓ **Classe avec de nombreux attributs/operations**

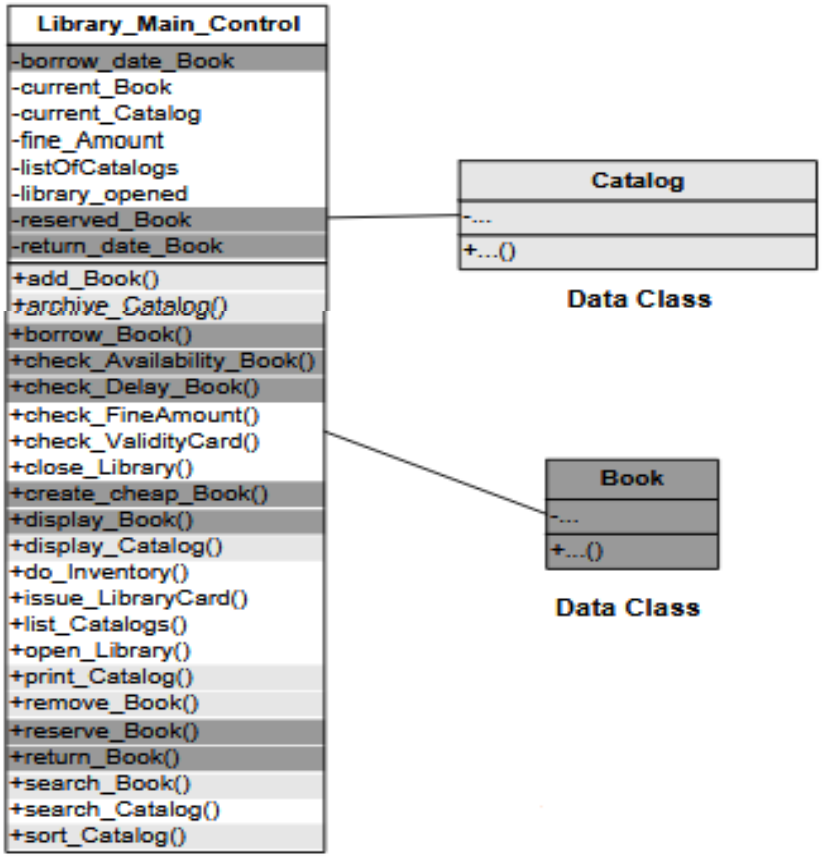
✓ **Perte des avantages lies a la POO;**

✓ **Objet trop complexe a réutiliser ou tester ;**

✓ **Objet lourd en terme d'occupation mémoire.**

Les anti-patrons

- **Blob, god object**



Les anti-patrons

- **Blob, god object**

Corriger un anti-patron Blob:

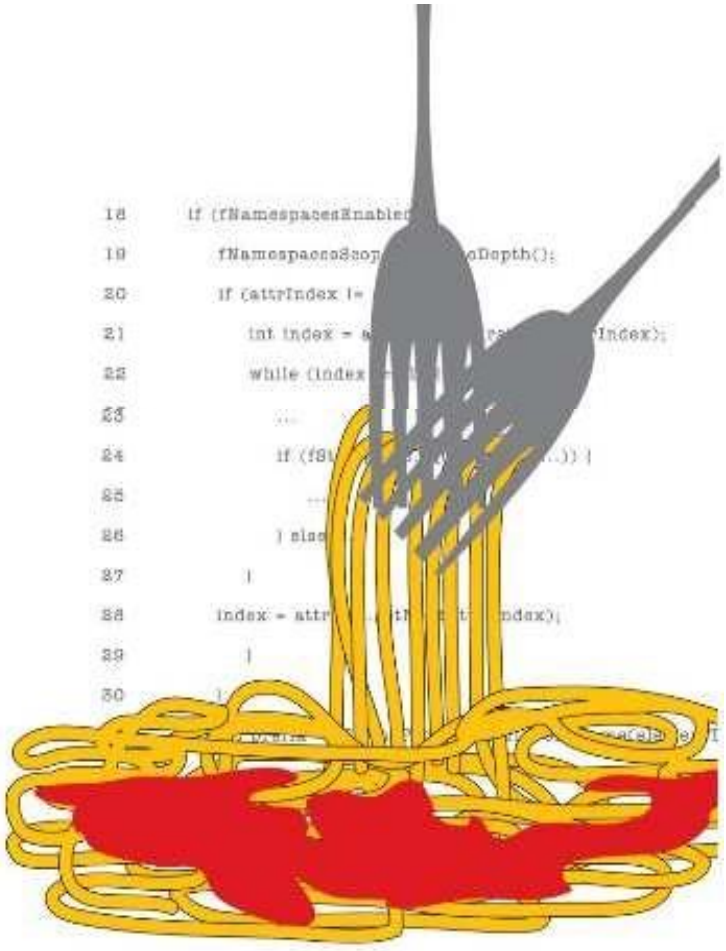
✓ identifier ou catégoriser les attributs et opérations liées.

✓Appliquer des techniques de conception objet (ex: héritage, délégation, patrons, etc.

Les anti-patrons

- **La programmation spaghetti**

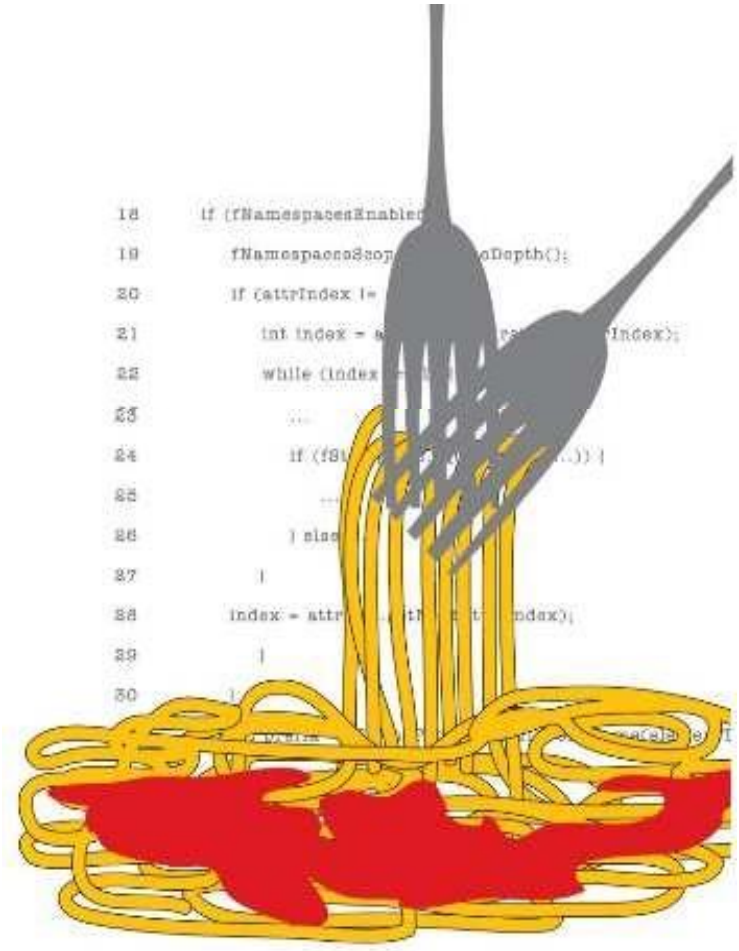
Ceci fait référence à l'image d'un plat de spaghetti, dans lequel il serait impossible de modifier une petite partie du logiciel sans altérer le fonctionnement de tous les autres composants.



Les anti-patrons

- **La programmation spaghetti**

Ceci fait référence à l'image d'un plat de spaghetti, dans lequel il serait impossible de modifier une petite partie du logiciel sans altérer le fonctionnement de tous les autres composants.



Les anti-patrons

- **La programmation spaghetti**

Impacts sur le code source :

- **Nombreux objets avec des opérations sans paramètres**
- **50% de la maintenance passée en redécouverte ;**
- **…**

Les anti-patrons

Le culte du cargo

❑**On parle de culte du cargo** lorsqu'un programmeur emprunte un bout de code sans le comprendre et espère qu'il fera la chose attendue dans un tout autre contexte.

❑**Programmation par copier-coller :** Consiste en copier quelque chose sans vraiment comprendre ou en ignorant les principes sous-jacents qui font que la solution originelle est opérationnelle mais inadaptée au problème pose.

Les anti-patrons

- **Le culte du cargo**

Origine de l'appelation : Culte réel ayant émergé en Melanesie âpres la seconde guerre mondiale.

Les anti-patrons

- **Le culte du cargo**

Un problème de ressources humaines plus que de code

La duplication de code sans vérification entraîne des incohérences. La meilleure solution étant encore de factoriser les parties communes au lieu de les dupliquer.