# 06 Chapitre

# Conclusion du cours

## 2I1AC3 : Génie logiciel et Patrons de conception

Régis Clouard, ENSICAEN - GREYC

# Patrons de conception



- Un patron de conception (Design pattern) est une solution éprouvée à un problème récurrent dans la conception d'applications orientée objet.
- Les patrons sont décrits au niveau conception et sont donc indépendants des langages de programmation utilisés.

# Patrons de conception de la bande des quatre (GOF)



- À leur création, on croyait que ce serait les premiers, mais il s'est avéré que ce sont pratiquement les seuls …
  - 23 (22) patrons

| Création | Structure | Comportement |
|---|---|---|
| | | ~~Interpréteur~~ |
| | | Patron de méthode |
| | Adaptateur | Chaîne de responsabilités |
| Fabrique simple | Pont | Commande |
| Fabrique abstraite | Composite | Itérateur |
| Monteur | Décorateur | Médiateur |
| Prototype | Façade | Mémento |
| Singleton | Poids mouche | Observateur |
| | Procuration | État |
| | | Stratégie |
| | | Visiteur |

# Objectif du cours



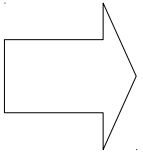
- Tous les patrons n'ont pas été présentés en cours, mais vous devez tous les connaître (sauf interpréteur).
- L'objectif du cours est surtout centré sur :
  - Appropriation des principes et des règles de conception avancés.
  - Sensibilisation à la qualité d'une conception : cohésion/couplage.
- A partir de ces principes et de ces règles, on peut retrouver tous les patrons.

# Principes de conception



- Votre code est STUPID rendez-le SOLID

**S** ingleton
**T** ight coupling
**U** ntestability
**P** remature optimization
**I** ndescriptive naming
**D** uplication



**S** ingle Responsibility
**O** pen-Closed
**L** iskov Substitution
**I** nterface Segregation
**D** ependency Inversion

# Le patron de conception Singleton



- Garantir qu'une classe ne possède qu'une seule instance.
- Principe de base :
  - Dérouter le constructeur et passer par une méthode statique.

```
public final class Singleton {
   private static Singleton _instance;        // 1
   private Singleton() { }                    // 2
   public static Singleton getInstance() {    // 3
      if (_instance == null) {
         _instance = new Singleton();
      }
      return _instance;
   }
}
```

  - Création d'une instance

```
Singleton s = Singleton.getInstance();
```

# L’anti-patron de conception Singleton



- Un singleton est revient à une variable globale avec les mêmes travers.
  - Instance globale au programme (difficile de revenir en arrière).
  - Utilisation par effets de bord.
  - Intestable : impossible de contrôler l’instanciation (p. ex. pas de doublure).
  - Concurrence : non partageable.
- Son utilisation doit être incontournable, une réponse à un problème critique, tel que :
  - La liste d'attente d'une imprimante 3D. Si la liste n'est pas partagée, il peut y avoir la fabrication d'objets bizarres.
  - La tour de contrôle d'un aéroport. S'il y a plusieurs instances, les avions ne sont pas tous connus par chaque instance et donc ils ne peuvent pas réellement régler le trafic.
  - Un robot médical pilotable par réseau accessible à plusieurs chirurgiens.
- Remarque : Le Singleton est très utilisé dans les framework tels Spring, mais c’est le compilateur qui les gère (par injection de dépendance).

# Règles de conception



- Règle 1. Réduire l'accessibilité des membres de classe.
  - Ne mettez les accesseurs que quand cela est exigé par le travail de développement.
  - Pensez vos classes comme des fournisseurs de service et pas comme des structures de données.
  - Objectif : réduire le couplage.

# Contre-exemple : patron Messager



- Un objet qui agit comme une simple structure
  - Ses données sont **publiques**.
- Quand ?
  - Les données et leur représentation sont la raison d'être de l'objet.
- Exemple 1
  - Un **Point 2D** est intrinsèquement caractérisé par ses deux attributs x et y. Donc les attributs font la classe et il n'y a pas de raison de les encapsuler.

```
public class Point2D {
    public int x;
    public int y;
}
```

- Exemple 2 :
  - Transformer une liste d'arguments d'une méthode par un seul objet.
  - f(a,b,c) → f(o) avec o.a, o.b, o.c

# Règles de conception



Règle 1. Réduire l'accessibilité des membres de classe.

Règle 2. Encapsuler les variations.

# Quiz : patron



- Quel patron de conception permet de capturer :
  - Les variations de comportement d'un **service** en fonction des **propriétés** d'un objet ?
    - Décorateur
  - Les variations sur des **services** dues à l'état de l'objet ?
    - État
  - Les variations de **parcours** d'un agrégat ?
    - Itérateur
  - Les variations de la liste des **objets dépendant** d'un objet de référence ?
    - Observateur
  - Les variations **d'algorithme** d'un même service ?
    - Stratégie
  - Les variations **d'une partie d'un algorithme** d'un service ?
    - Patron de méthode
  - Les variations dans la **liste des méthodes** d'une classe ?
    - Visiteur

# Règles de conception



Règle 1. Réduire l'accessibilité des membres de classe.

Règle 2. Encapsuler les variations.

Règle 3. Programmer pour une interface et non pour une implémentation.

- On remarque ainsi qu'il y a toujours une classe abstraite ou interface à la racine du graphe d'héritage des patrons.
- Le code doit être le plus générique possible en référençant les objets avec la classe la plus haute possible dans la hiérarchie »
  - p.ex. List<T> object  new ArrayList<>() ;

# Règles de conception



Règle 1. Réduire l'accessibilité des membres de classe.

Règle 2. Encapsuler les variations.

Règle 3. Programmer pour une interface et non pour une implémentation.

Règle 4. Privilégier la composition à l'héritage.

- Transformer une création d'objet statique (utilisant l'héritage) par une construction dynamique (à l'exécution).

# Quiz : patron

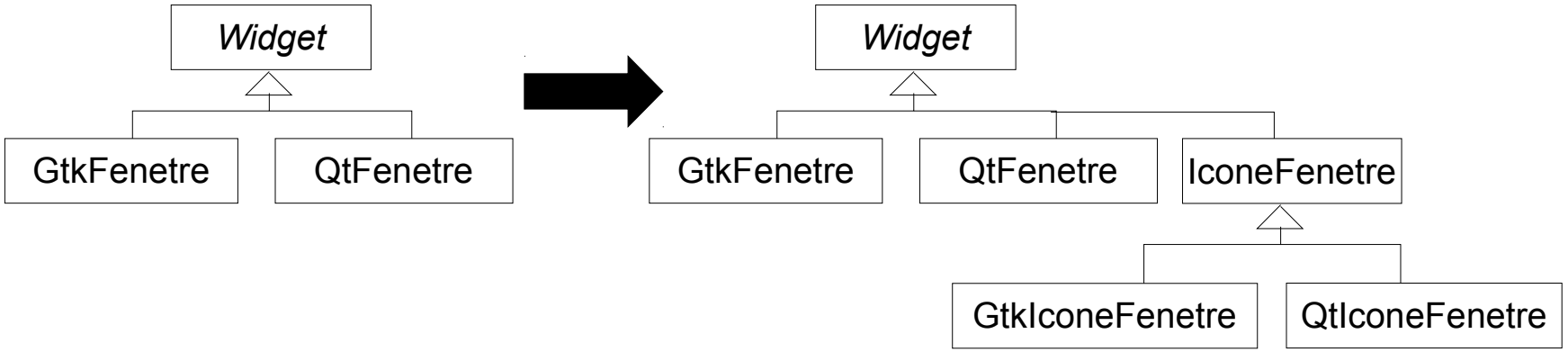


- Quel patron permet d'éviter de créer autant de classes que de combinaison de propriétés d'un objet ?
  - Décorateur
- Quel patron permet d'éviter de créer autant de classes que de combinaison d'abstraction et de d'implémentation ?
  - Pont

# Patron de conception Pont



- Exemple



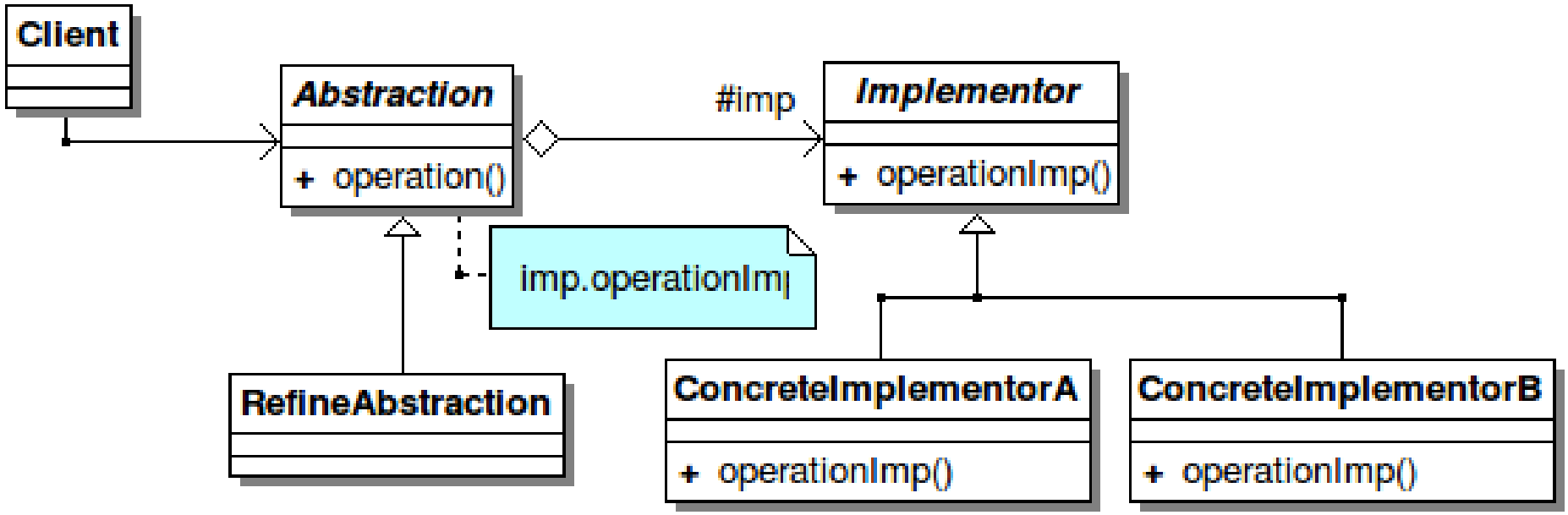

- Solution


- Construction d'une instance :
  - Widget wFenetreGtk = new Fenetre(new Gtk()) ;

# Patron de conception Pont



- Découpler les variations en abstraction et en implémentation pour éviter de créer des classes par leur combinaison.
- Puis les réunir par une association (pont)



# Révision des 22 patrons

# 1. Patrons de création



- Encapsulation de la création de classes ou d'objets.
  - Décrire la manière dont un objet ou un ensemble d'objets peuvent être créés, initialisés, et configurés.
  - Isoler le code relatif à la création et à l'initialisation afin de rendre l'application indépendante de ces aspects.
- Patrons :
  - **Méthode fabrique**
  - **Fabrique abstraite**
  - Monteur
  - Prototype
  - **Singleton**

# Patrons de création



- Méthode fabrique
  - Créer des objets sans spécifier sa classe exacte.
  - Isoler la création d'objet à un seul endroit.
  - *Exemple : chaque franchise d'une chaîne de pizzeria est déléguée à une sous-classe spécifique.*
- Fabrique abstraite
  - Créer des familles d'objets.
  - *Exemple : chaîne de fabrication de voiture.*

# Patrons de création



- Singleton (aussi un anti-patron !)
  - Garantir une seule instance d'une classe.
  - *Exemple : une bibliothèque graphique.*
- Monteur
  - Séparer le processus de construction de l'objet de sa représentation finale.
  - Le processus de construction est identique mais le produit fini peut varier.
  - *Exemple : création d'un menu de fast-food.*
- Prototype
  - Spécifier les espèces d'objet à créer en utilisant une instance prototypique, et créer de nouveaux objets par copie de ce prototype.
  - *Exemple : réplication de l'ADN*

# 2. Patrons de structure



- Abstraction de la composition de structures de classes ou d'objets plus importantes.
  - Décrire la manière dont des objets de l'application doivent être connectés afin de rendre ces connexions indépendantes des évolutions futures de l'application.
  - Découpler l'interface de l'implémentation de classes et d'objets.
- Patrons :
  - **Adaptateur**
  - **Pont**
  - **Décorateur**
  - **Procuration**
  - **Composite**
  - Façade
  - Poids mouche

# Patrons de structure



- Adaptateur
  - Convertir l'interface d'une classe pour la conformer à l'attente de l'utilisateur.
  - *Exemple : contrôle d'appareil électrique.*
- Pont
  - Découpler une abstraction de son implémentation associée afin que les deux puissent évoluer indépendamment.
  - *Exemple : bibliothèques graphiques.*
- Décorateur
  - Attacher des responsabilités supplémentaires à un objet de façon dynamique.
  - *Exemple : combattants dans un jeu de rôle.*

# Patrons de structure



- Composite
  - Organiser les objets en structure arborescente représentant la hiérarchie de bas en haut.
  - Permet aux utilisateurs de traiter des objets individuels et des ensembles organisés de ces objets de la même façon.
  - *Exemple : langage ensembliste de formes.*
- Procuration
  - Fournir un subrogé ou un remplaçant d'un objet pour en contrôler l'accès.
  - Exemple : Image dans un traitement de texte.

# Patrons de structure



- Façade
  - Fournir une interface qui rend un sous-système plus facile à utiliser.
- Poids mouche
  - Assurer en mode partagé le support d'un grand nombre d'objets à fine granularité. Un poids mouche est un objet partagé susceptible d'être utilisé simultanément dans plusieurs contextes.

# 3. Patrons de comportement



- Interaction de structures d'objets ou de classes
  - Décrire le comportements d'interaction entre objets
  - Gérer les interactions dynamiques entre des classes et des objets.
- Patrons
  - Patrons de méthode
  - Chaîne de responsabilités
  - **Commande**
  - **Itérateur**
  - Médiateur
  - Mémento
  - **Observateur**
  - **État**
  - **Stratégie**
  - **Visiteur**

# Patrons de comportement



- Observateur
  - Définir une corrélation entre objets de type un à plusieurs de façon à ce que lorsqu'un objet change d'état tous ceux qui en dépendent en soient notifiés.
  - *Exemple : les graphiques d'un tableur.*
- Stratégie
  - Définir une famille d'algorithmes, encapsuler chacun d'eux et les rendre interchangeables
  - Modifier un algorithme d'un service indépendamment de ses clients.
  - *Exemple : tri d'un tableau de données.*

# Patrons de comportement



- État
  - Modifier le comportement d'un objet lorsque son état interne change.
  - L'objet paraîtra changer de classe.
  - *Exemple : la conte de la grenouille et du prince.*
- Commande
  - Encapsuler l'invocation d'un service dans un objet à part entière.
  - *Exemples : mécanisme de reversion, historique des commandes.*

# Patrons de comportement



- Itérateur
  - Fournir un moyen pour accéder en séquence aux éléments d'un objet de type agrégat sans révéler sa représentation interne.
  - *Exemple : parcours d'un forme complexe*
- Visiteur
  - Ajouter une nouvelle opération sans modifier les classes des éléments sur lesquelles elle opère.
  - *Exemple: Plugins sur la structure d'objet du langage ensembliste.*

# Patrons de comportement

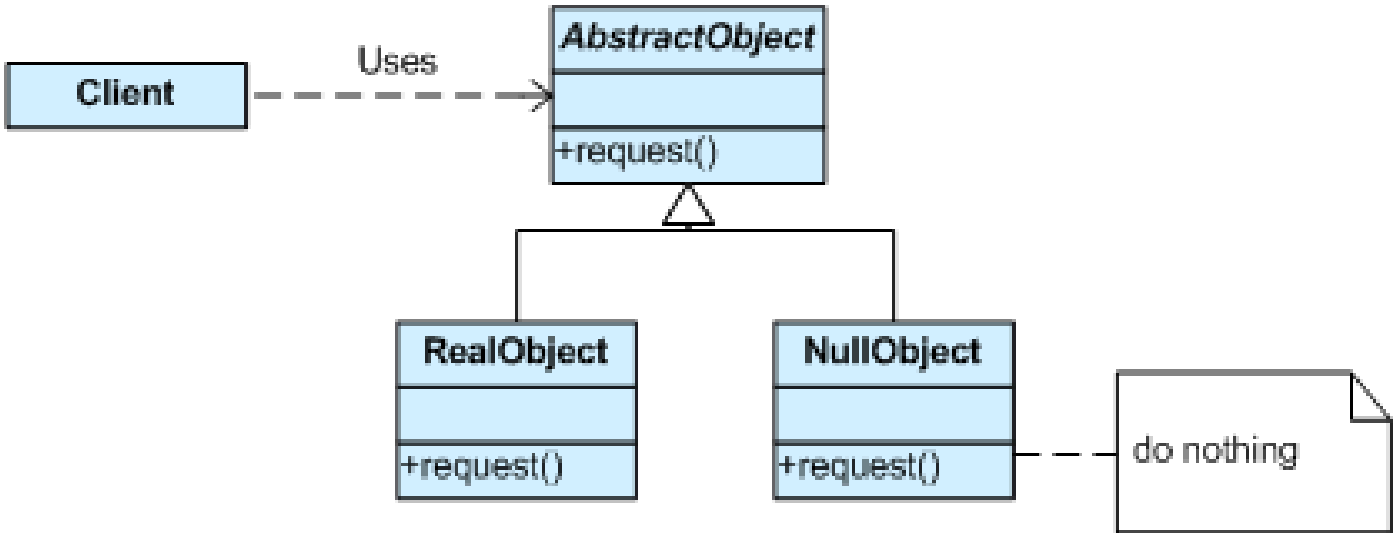


- Chaîne de responsabilité
  - Définir une chaîne d'objet susceptibles de répondre à une requête sans connaître les possibilités des objets sur cette requête.
- Médiateur
  - Régler le problème de communication entre des objets qui doivent s'ignorer.
- Memento
  - Stocker les données d'un objet en mode protégé.
- Patron de méthode
  - Pouvoir varier les parties d'un algorithme.

# D’autres patrons (non GOF)



- Objet nul
  - Éviter les if (o != null) { o.method(); }



# Anti-patrons

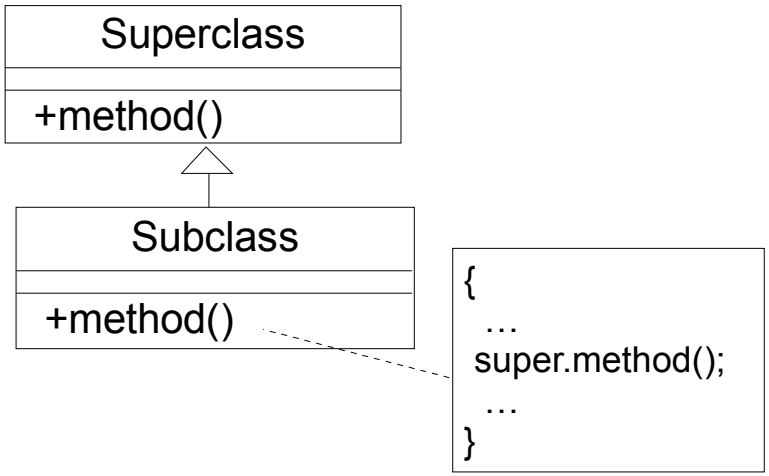


- Les anti-patrons sont des erreurs courantes de conception des logiciels.
- Les anti-patrons se caractérisent souvent par une lenteur excessive du logiciel, des coûts de réalisation et de maintenance élevés, des comportements anormaux et l'introduction de bugs.

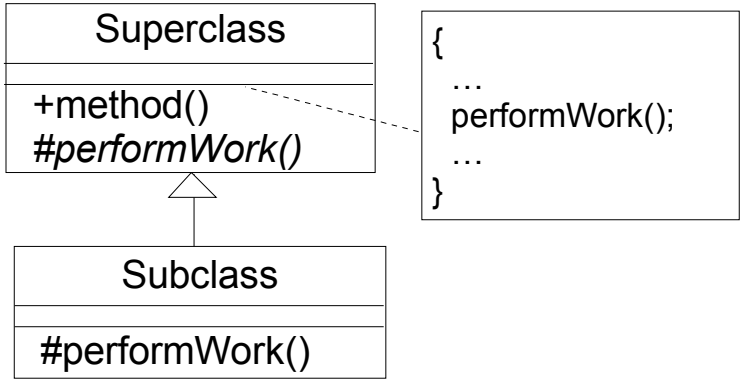
# Anti-pattern



- Call super
  - Mettre une méthode par défaut dans la classe abstraite sans obliger les sous-classes à redéfinir la méthode.




- Solution : patron Template Method




- *Remarque : pour résoudre ce problème Android utilise l'annotation @CallSuper sur les méthodes.*

# Conclusion



- Patrons de conception
  - Les patrons ont quelques fois des coûts supplémentaires.
  - Par contre d'autres sont des solutions sine qua none (adaptateur, commande, décorateur, visiteur…).
- Pour chaque règle, il existe des cas où son application serait une pure folie.
  - e.g raison de sécurité, de performances, etc.
- Contrairement à une première idée, les patrons de conception se marient très bien avec les approches Agiles.
  - Le principe YAGNI suggère de commencer par la solution la plus simple n'intégrant pas forcément de patrons de conception.
  - Ensuite, le développement itératif incite à se poser la question d'une refonte de la conception reposant sur les patrons pour prendre en compte de nouvelles fonctionnalités réutilisant des parties du code existant.