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.
- Un patron est décrit par :
  - nom
  - problème
  - solution
  - conséquences

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

		Role		
		Creation	**Structure**	**Behavioral**
Domain	**Class**	Factory method	Adapter	~~Interpreter~~ Factory Method
	Object	**Abstract Factory** Builder Prototype Singleton	Adapter Bridge Composite Decorator **Facade** **Flyweight** Proxy	**Chain of responsibility** Command Iterator **Mediator** **Memento** Observer State Strategy Visitor

Objectif du cours

- Tous les patrons n'ont pas été présentés en cours, mais vous devez les apprendre tous (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

Règles de conception

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

Contre-exemple : Messager

- Un objet qui agit comme une simple structure
  - Données sont **publiques**
- Quand ?
  - L'objet est naturellement caractérisé par ses données.
  - On veut simplifier la liste d'arguments de méthodes.
- Exemple
  - 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;
}
```

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 les services dues au type de l'objet ?
    - État
  - Les variations de parcours d'un agrégat ?
    - Itérateur

Quiz : patron

- 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 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.

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.

Quiz : patron

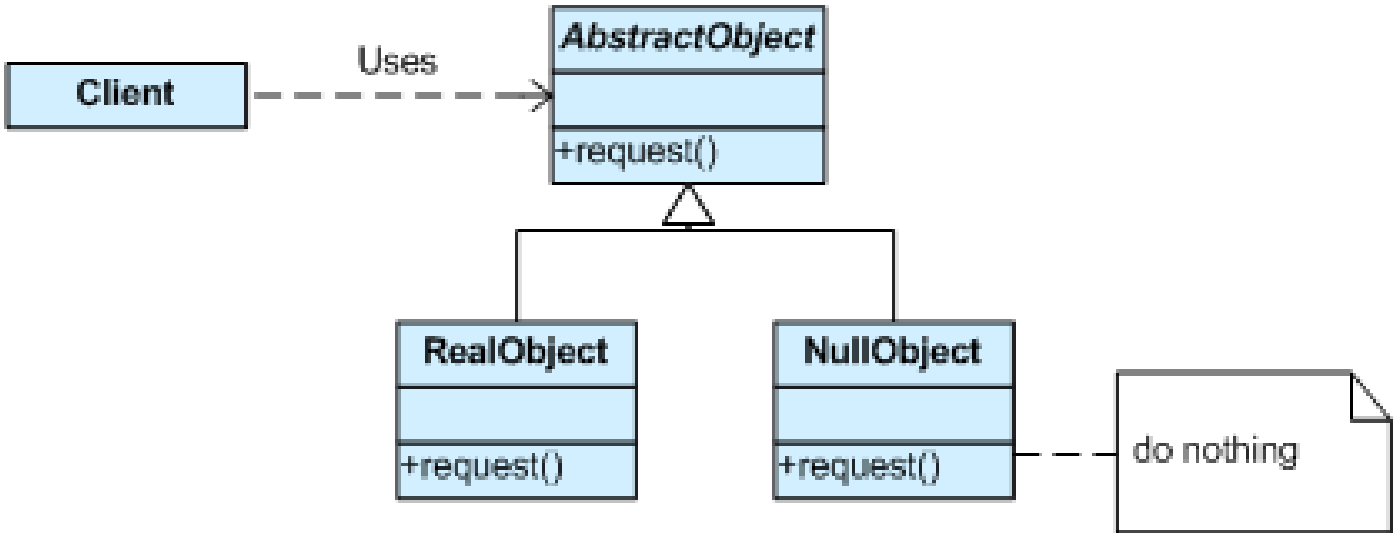
- Quel patron permet d'éviter de créer autant de classes que de combinaison d'abstraction et de d'implémentation ?
  - Pont
- Quel patron permet d'éviter de créer autant de classes que de combinaison de propriétés ?
  - Décorateur

NullPointerException

- Comment éviter les NullPointerException
  - Ne jamais utiliser de null dans le code.
  - Lancer des exceptions en cas d'erreur.
  - Utiliser les *Optional* :
    - ```
      if (myObject != null) {
          myObject.myMethod();
      }
      ```
    - ```
      if (myObjectOptional.isPresent()) {
          myObjectOptional.get().myMethod();
      }
      ```
    - Avantage, le client est informé et obligé de traiter le cas null.
  - Utiliser le patron de conception « Objet Nul ».

Objet nul

- Éviter les o == null

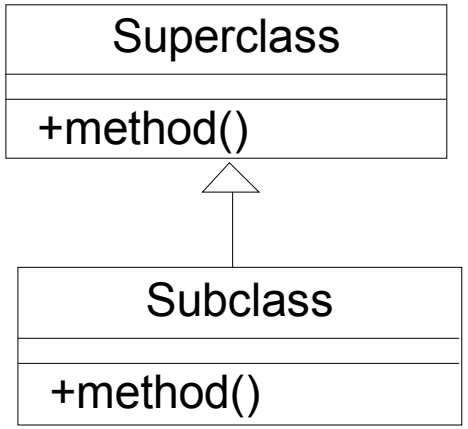


Anti-patrons

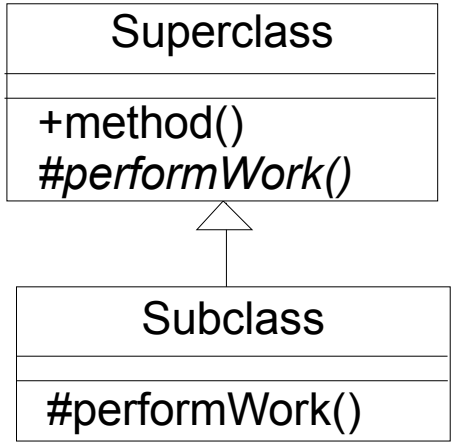
- Les anti-patrons sont des erreurs courantes de conception des logiciels.
- Les anti-patrons se caractérisent souvent par une lenteur excessive du logiciel, des coûts de réalisation et de maintenance élevés, des comportements anormaux et la présence de bugs.

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'un refactoring de la conception reposant sur les patrons pour prendre en compte de nouvelles fonctionnalités réutilisant des parties du code existant.

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, l'initialisation afin de rendre l'application indépendante de ces aspects.
- Patrons :
  - Motif de création de classe (héritage)
    - **Méthode fabrique**
  - Motif de création d'objets (délégation)
    - **Singleton, Monteur**

Patrons de conception

- Méthode fabrique
  - Créer des objets sans spécifier sa classe exacte.
  - Isoler la création d'objet à un seul endroit.
  - Exemple : pizzeria. Chaque franchise est déléguée à une sous-classe spécifique.
- 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.

Patrons de structure

- Abstraction de la composition de structures de classes ou d'objets plus importantes.
  - Décrire la manière dont doivent être connectés des objets de l'application afin de rendre ces connexions indépendantes des évolutions futures de l'application.
  - Découpler l'interface de l'implémentation de classes et d'objets.
- Patrons :
  - Structure de classes
    - Utilisation de l'héritage : **Adaptateur**
  - Composition d'objets
    - Ajout d'un niveau d'indirection : **Adaptateur**, **Pont**, **Décorateur**, **Procuration**
    - Composition récursive : **Composite**

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 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
  - Dépendance entre objets : **Observateur**
  - Délégation de services : **Stratégie**, **État**, **Commande**
  - Parcours de structures : **Itérateur**, **Visiteur**

Patrons de conception

- Observateur
  - Définir une corrélation entre objets de type un à plusieurs de façon que lorsqu'un objet change d'état tous ceux qui en dépendent en soient notifiés et mis à jour automatiquement.
  - *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 conception

- 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 une requête comme un objet ce qui permet de faire un paramétrage des clients avec différentes requêtes, files d'attentes ou historiques de requêtes et d'assurer le traitement des opérations réversibles.
  - *Exemple : undo / redo*

Patrons de conception

- 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.*