ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Résumé du cours

2I1AC3 : Génie logiciel et Patrons de conception

Régis Clouard, ENSICAEN - GREYC

Patrons de conception
Design patterns

- Un patron de conception est une solution éprouvée à un problème récurrent dans la conception d'applications orientées 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 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 connections 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

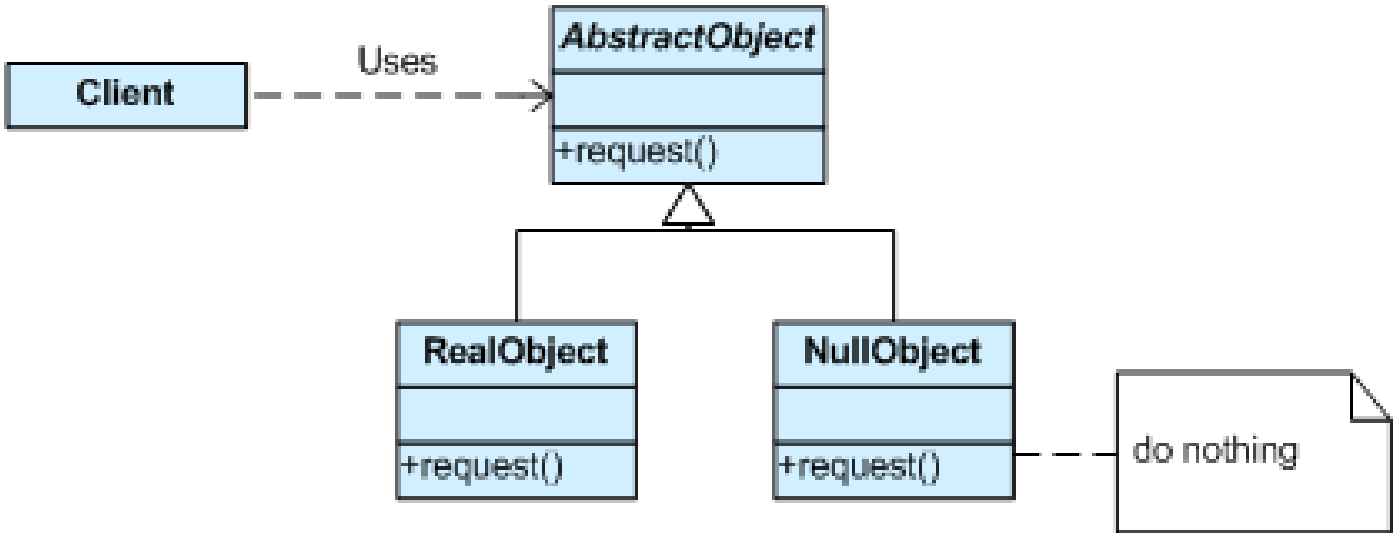
- Itérateur
  - *Fournir un moyen pour accéder en séquence aux éléments d'un objet de type agrégat sans révéler sa représentation interne.*
  - *Exemple : parcours d'un forme complexe*
- *Visiteur*
  - Ajouter une nouvelle opération sans modifier les classes des éléments sur lesquelles elle opère.
 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

Tous les patrons de conception

		Role		
		Creation	**Structure**	**Comportement**
Domaine	**Classe**	*Méthode fabrique*	Adaptateur	~~Interpreteur~~ **Patron de méthode**
	Objet	**Fabrique abstraite** **Monteur** **Prototype** *Singleton*	Adaptateur Pont Composite Décorateur **Facade** **Poids mouche** Procuration	**Chaîne de responsabilité** *Commande* *Etat* *Iterateur* **Médiateur** **Memento** *Observeur* *Stratégie* *Visiteur*

L'objet nul

- Éviter les o == null



NullPointerException

- Comment éviter les NullPointerException
  - Ne jamais utiliser de null dans le code
  - Utiliser le patron « objet nul »
  - Lancer des exceptions en cas d'erreur.
  - Utiliser les Optional (indiquer qu'une retour de méthode peut potentiellement retourner une valeur vide (pas nulle)).
    - Exemple sans Optional

```
if (myObject != null) {
    myObject.myMethod();
}
```

    - Exemple avec Optional

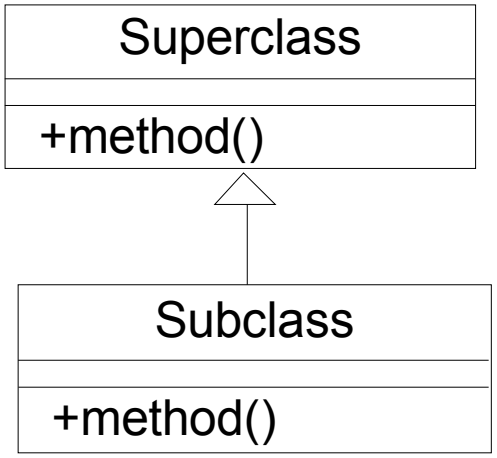
```
if (myObjectOptional.isPresent()) {
    myObjectOptional.get().myMethod();
}
```

Anti-patons

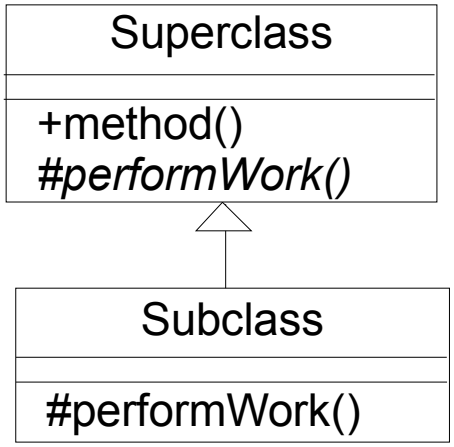
- Les anti-patrons sont des erreurs courantes de conception des logiciels.
- Les anti-patrons se caractérisent souvent par une lenteur excessive du logiciel, des coûts de réalisation et de maintenance élevés, des comportements anormaux et la présence de bugs.

Anti-pattern

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



- Solution



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

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
  - Une instance de la classe **Point** mieux que les deux entiers x et y

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

Règles de conception

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

Règle 2. Encapsuler ce qui varie.

Quiz : patron ↔ variation

- 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 (suite)

- 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 ce qui varie.

Règle 3. Programmer pour une interface pas pour une implémentation.

Règles de conception

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

Règle 2. Encapsuler ce qui varie.

Règle 3. Programmer pour une interface pas pour une implémentation.

Règle 4. Préférer la composition à l'héritage.

Quiz : patron ↔ composition vs héritage

- Quel patrons 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

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, …).
- Pour chaque règle, il existe des cas où appliquer la patron serait une pure folie.
- Contrairement à une première idée, les patrons de conception se marient très bien avec les approches Agiles.
  - En effet, c'est lors d'un « refactoring » que l'on peut s'apercevoir qu'il devient intéressant d'utiliser un patron.