ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

03
Chapitre

Conclusion sur les patrons de conceptions

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 orientée objet de logiciels.
- 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 généraux…
  - 23 patrons (18 sont considérés dans ce cours)

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

  - À connaître par cœur : ils forment le socle culturel du génie logiciel

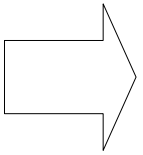
Objectif du cours

- 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.

1. Principes de conception

- Votre code est STUPID rendez-le SOLID

S ingleton
T
U
P
I
D



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 **jamais** plus d'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 synchronized Singleton getInstance() { // 
3
      if (_instance == null) {
         _instance = new Singleton();
      }
      return _instance;
   }
}
```

  - Création d'une instance

```
Singleton s = new Singleton();
```

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

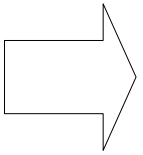
L’anti-patron de conception Singleton

- Un singleton est comparable à une variable globale avec les mêmes travers.
  - Instance globale au programme : code rigide.
  - Utilisation par effets de bord : incontrôlable.
  - Concurrence : non partageable.
  - Intestable : impossible de doubler l’instance.
- Son utilisation doit être incontournable, une réponse à un problème critique, tel que :
  - La liste d'attente d'une imprimante 3D (en supposant qu’il ne peut y avoir qu’une imprimante dans le système d’information). Si la liste n'est pas partagée, il peut y avoir la fabrication d'objets bizarres.
  - La tour de contrôle d'un aéroport.
  - Un robot médical dans une maison médicale en campagne pilotable par réseau accessible à plusieurs chirurgiens.

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

2. Règles de conception

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

Règle 2. Encapsuler ce qui varie dans une hiérarchie spécifique.

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

Règle 4. Privilégier la composition à l'héritage.

2. Règles de conception

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

- Ne mettez les accesseurs / mutateurs 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 (tight coupling).

Règle 2. Encapsuler ce qui varie dans une hiérarchie spécifique.

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

Règle 4. Privilégier la composition à l'héritage.

Contre-exemple : le patron Messager

- Un objet qui agit comme une simple structure de données
  - 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
    - Permet d'augmenter le nombre de paramètres sans modifier la signature de la méthode (→ natif dans les langages Kotlin, Python ou Dart).

Règles de conception

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

Règle 2. Encapsuler ce qui varie dans une hiérarchie spécifique.

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

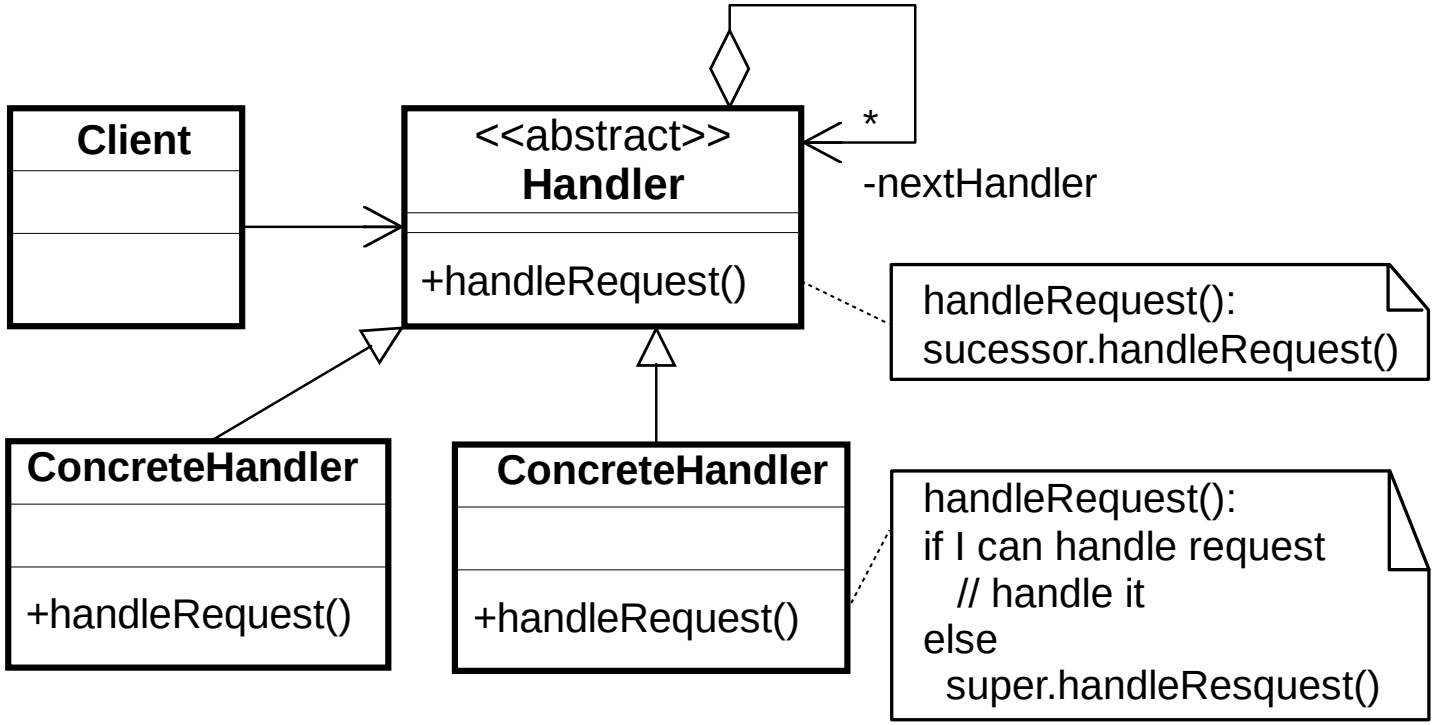
Règle 4. Privilégier la composition à l'héritage.

Quiz : patron

- Quel patron de conception permet de capturer les variations :
  - variation de comportement **d'un service** d'un objet en fonction de ses **propriétés** ?
    - Décorateur
  - variation du comportement **des services** d'un objet en fonction de son état ?
    - État
  - variation de **parcours** d'un agrégat ?
    - Itérateur
  - variation de la liste des **objets dépendant** d'un objet de référence ?
    - Observateur
  - variation de **l'algorithme** d'un service ?
    - Stratégie
  - variation **d'une partie de l'algorithme** d'un service ?
    - Patron de méthode
  - variation de la **liste des méthodes** d'une classe ?
    - Visiteur
  - variation sur l'objet d'une chaîne capable de répondre à requête
    - Chaîne de responsabilité

Le patron chaîne de responsabilité

- **Quand ?**
  - Déléguer la réalisation d'une fonctionnalité à une chaîne de délégation inconnue (cf. le tondeur de pelouse).
- **Ce qui varie**
  - L'objet qui peut répondre à un service
- **Solution**



Règles de conception

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

Règle 2. Encapsuler ce qui varie dans une hiérarchie spécifique.

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<>();`
- Il est ainsi moins dépendant des évolutions qui interviennent le plus souvent sur les implémentations.

Règle 4. Privilégier la composition à l'héritage.

Règles de conception

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

Règle 2. Encapsuler ce qui varie dans une hiérarchie spécifique.

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

Règle 4. Privilégier la composition à l'héritage.

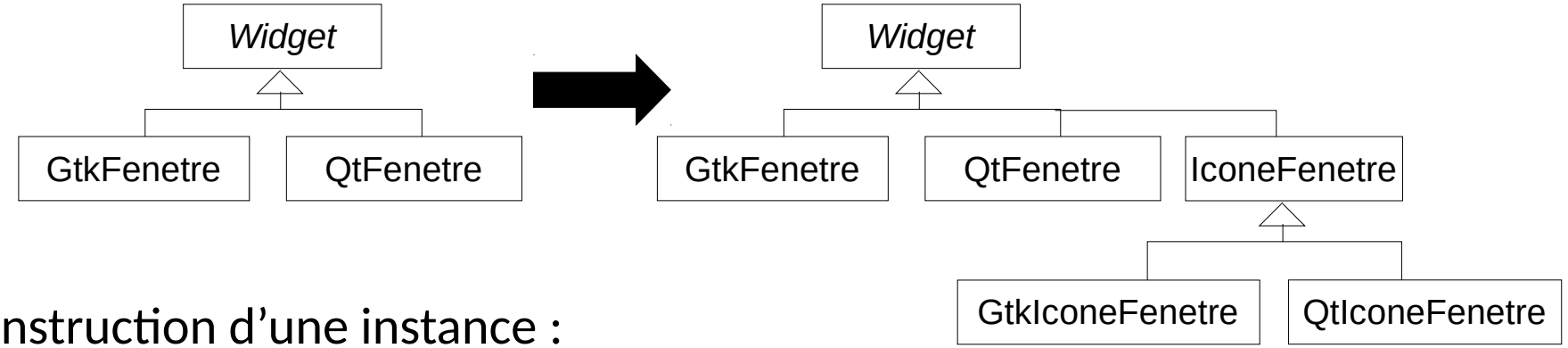
- L'héritage est très souvent un erreur de conception.
- Il faut transformer la création d'objet composite statique (utilisant l'héritage) par la construction dynamique (à l'exécution).
  - ie `new Classe1Classe2() vs new Classe1(new Classe2());`
- Ne jamais dériver de classes concrètes : risque de violer le principe de substitution de Liskow

Quiz : patron

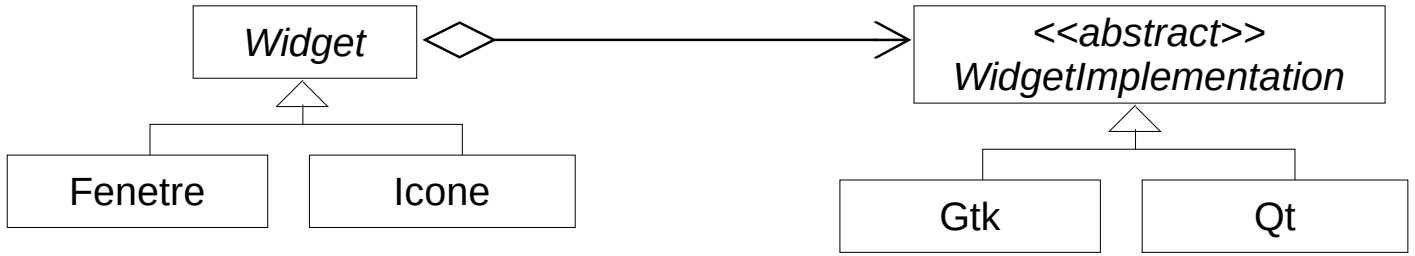
- Quel patron permet d'éviter de créer autant de classes que de combinaison de propriétés d'un objet ?
  - Décorateur
- Quel patron permet d'éviter de créer autant de classes que de combinaison d'abstraction et d'implémentation ?
  - Pont

Patron de conception Pont

- Exemple



  - Construction d'une instance :
    - Widget fenetreGtk = new GtkFenetre();
- Solution : Découpler les variations en abstraction et en implémentation pour éviter de créer des classes par leur combinaison.



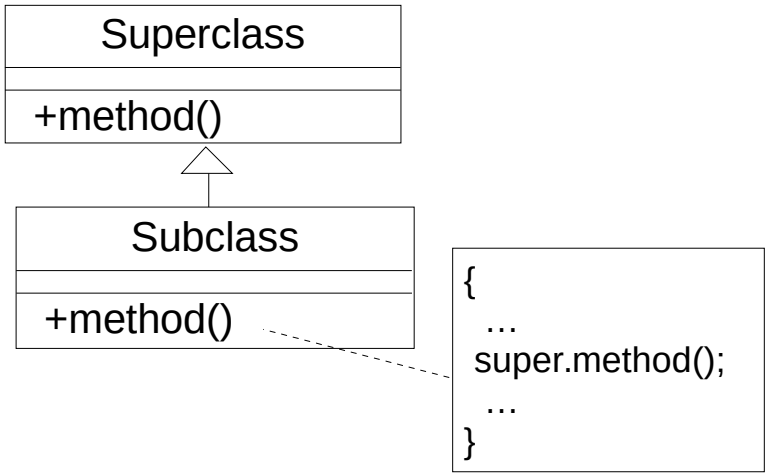
  - Construction d'une instance :
    - Widget fenetreGtk = new Fenetre(new Gtk());

Anti-patrons

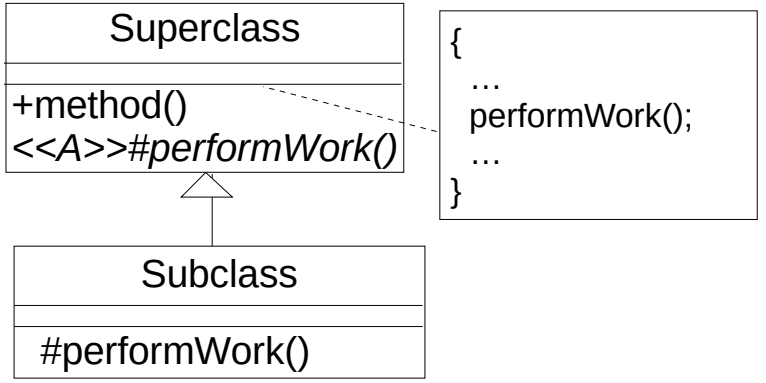
- Les anti-patrons sont des erreurs courantes dans la conception des logiciels.
- Les anti-patrons se caractérisent souvent par une lenteur excessive du logiciel, des coûts de réalisation et de maintenance élevés, des comportements anormaux et l'introduction de bugs.

Anti-pattern

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



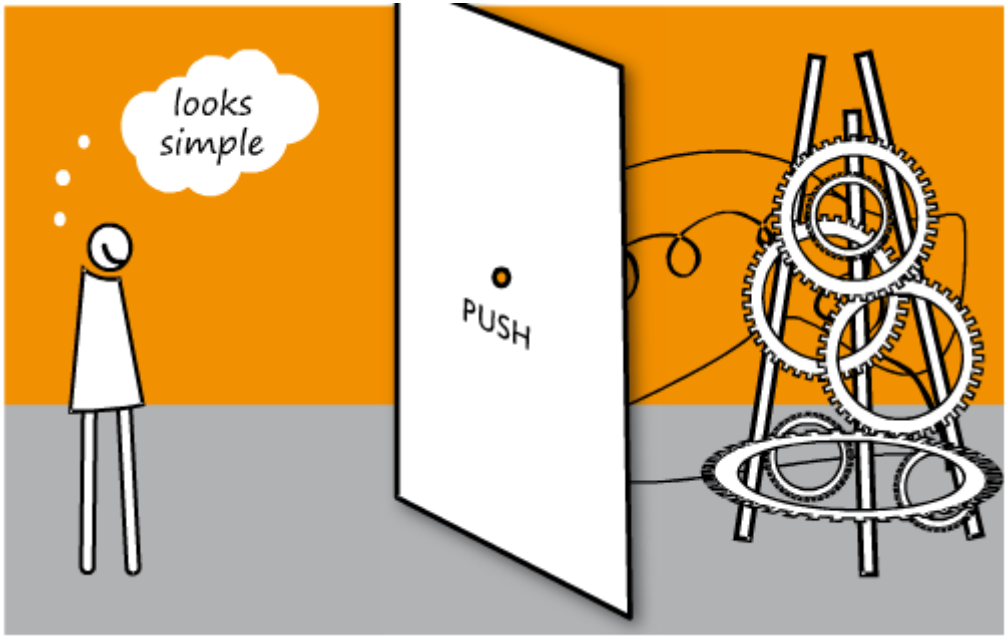
- Solution : patron Template Method



- *Remarque : pour résoudre ce problème Android utilise l'annotation @CallSuper sur les méthodes (eg, onCreate, onResume, onDestroy)*

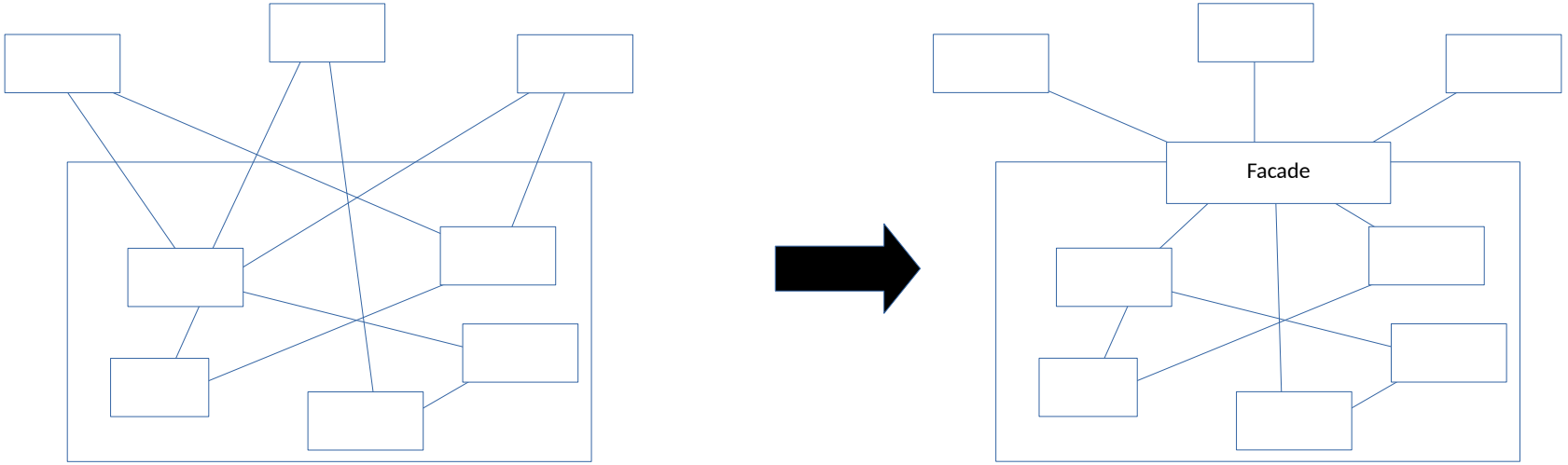
Le patron Façade

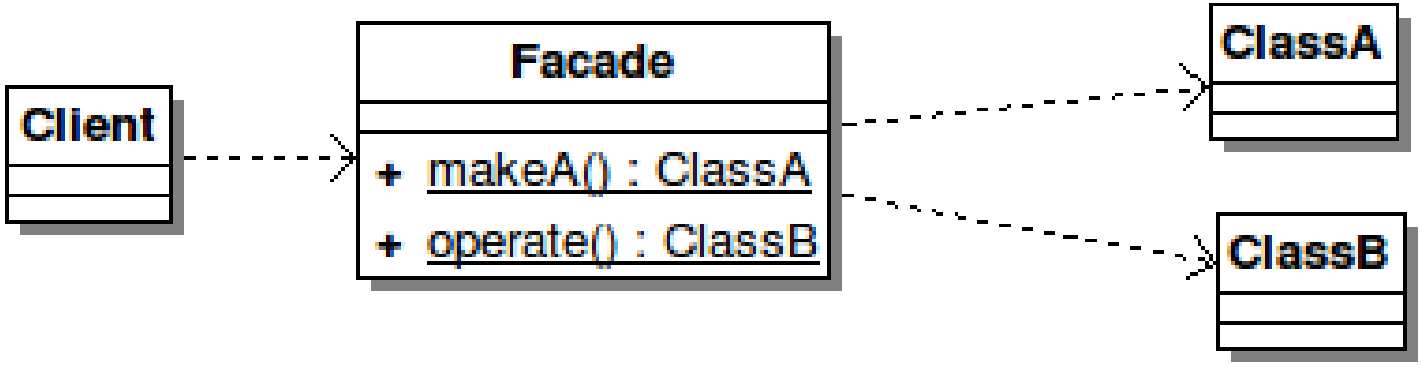
- Fournir une interface qui rend un sous-système plus facile à utiliser.



Le patron Façade

- **Quand** ?
  - On souhaite simplifier l'interface d'accès à un sous-système
- **Solution**
  - Une classe abstraite qui regroupe les éléments de l'interface.
  - Cf. Principe de ségrégation des interfaces.





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…).
- Attention : 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.
    - Ne pas accumuler de dettes techniques.
  - Recherche de l'excellence technique.

Révision des 18 principaux patrons
(non faits : Prototype,
Médiateur, Mémento,
Poids mouche, Interpréteur)

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 : fabrique de combattants : commandant et soldat*
- Fabrique abstraite
  - Créer des familles d'objets.
  - *Exemple : de fabrication de combattant Extraterrestre et Humain*

Patrons de création

- Singleton (aussi un anti-patron !)
  - Garantir une seule instance d'une classe.
  - *Exemple : un gestionnaire d'impression (si un seul !)*
- 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 : réponse à l'anti-patron constructeur télescopique*

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 avec des pouvoirs et des qualités optionnels.*

Patrons de structure

- Composite
  - Organiser les objets en structure arborescente.
  - 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.

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
  - ~~Interpréteur~~
  - 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 d'opérations de calcul 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.
- Patron de méthode
  - Pouvoir varier les parties d'un algorithme.

D’autres patrons (non GOF)

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