Design Patterns

Récapitulatif des informations issues du livre Design Patterns de la collection Tête la Première des éditions O'Reilly. Il est basé sur le langage Java.

Définition d'un Design Pattern

Un **pattern** est une solution à un problème dans un contexte.

Définition d'un Anti-Pattern

- Un **Anti-Pattern** vous dit comment partir d'un problème et parvenir à une mauvaise solution.
- Un anti-pattern vous dit pourquoi une mauvaise solution est attrayante.
- Un anti-pattern vous dit pourquoi cette solution est mauvaise à long terme.
- Un anti-pattern suggère d'autres patterns applicables pouvant fournir de meilleures solutions.

Conseils importants

Pour modifier le comportement d'un élément (ex : un canard) au moment de l'exécution, il suffit d'appeler la méthode set() correspondant à ce comportement.

Designs Patterns

Pattern Stratégie (Strategy)

Le pattern Stratégie définit une famille d'algorithmes (interface avec des sous-classes), encapsule chacun d'eux et les rend interchangeables. Stratégie permet à l'algorithme de varier indépendamment des clients (ex : des Canards, des voitures) qui l'utilisent. Encapsule des comportements interchangeables et emploie la délégation pour décider lequel utiliser.

Pattern Observateur (Observer)

Le pattern Observateur définit une relation entre objets de type un-à-plusieurs, de façon que, lorsque un objet change d'état, tous ceux qui en dépendent en soient notifiés et soient mis à jour automatiquement (ex : abonnement à un journal. Le sujet est l'éditeur et les abonnés sont les observateurs. Quand les données du sujet (le journal) change, les observateurs (abonnées) en sont informés). Le « un » est le sujet, et le « plusieurs » constitue les éléments qui affichent les éléments.

Pattern Décorateur (Decorator)

Le pattern Décorateur attache des responsabilités supplémentaires à un objet de façon dynamique. Il permet une solution alternative pratique à la dérivation pour étendre les fonctionnalités.

Pattern Fabrication (Factory Method)

Le pattern Fabrication définit une interface pour la création d'un objet, mais en laissant aux sous-classes le choix des classes à instancier. Fabrication permet à une classe de déléguer l'instanciation à des sous-classes. La Fabrique se **base sur l'héritage**.

Pattern Fabrique Abstraite (Abstract Factory)

Le pattern Fabrique Abstraite fournit une interface pour créer des familles d'objets apparentés ou dépendants sans avoir à spécifier leurs classes concrètes. La Fabrique Abstraite se **base sur la composition**.

Pattern Singleton (Singleton)

Le pattern Singleton garantit qu'une classe n'a qu'une seule instance et fournit un point d'accès global à cette instance.

Pattern Commande (Command)

Le pattern Commande encapsule une requête comme un objet, autorisant ainsi le paramétrage des clients par différentes requêtes, files d'attente et récapitulatifs de requêtes, et de plus, permettant la reversibilité des opérations.

Pattern Adaptateur (Adapter)

Le pattern Adaptateur convertit l'interface d'une classe en une autre conforme à celle du client. L'Adaptateur permet à des classes de collaborer, alors qu'elles n'auraient pas pu le faire du fait d'interfaces incompatibles. Prendre l'exemple de l'adaptateur de la prise de courant anglaise pour la prise européenne. Il existe **l'Adaptateur de classe** basé sur l'héritage multiple et **l'Adaptateur d'objet** basé sur la composition.

Pattern Façade (Facade)

Le pattern Façade fournit une interface unifiée à l'ensemble des interfaces d'un sous-système. La façade fournit une interface de plus haut niveau qui rend le sous-système plus facile à utiliser.

Pattern Patron de méthode (Template Method)

Le pattern Patron de méthode définit le squelette d'un algorithme dans une méthode, en déléguant certaines étapes aux sous-classes. Patron de méthode permet aux sous-classes de redéfinir certaines étapes d'un algorithme sans modifier la structure de celui-ci.

Pattern Itérateur (Iterator)

Le pattern Itérateur fournit un moyen d'accéder en séquence à un objet de type collection sans révéler sa représentation sous-jacente. Il attribue également cette tâche de navigation à l'objet itérateur, non à la collection, ce qui simplifie l'interface et l'implémentation de la collection et place la responsabilité au bon endroit. Un itérateur a pour tâche de parcourir une collection et encapsule l'itération dans un autre objet.

Pattern Composite (Composite)

Le pattern Composite compose des objets en des structures arborescentes pour représenter des hiérarchies composant/composé. Il permet aux clients de traiter de la même façon les objets individuels et les combinaisons de ceux-ci. Avec une structure composite, nous pouvons appliquer les mêmes opérations aux composites et aux composants. En d'autres termes, nous pouvons **ignorer** dans la plupart des cas les différences entre les compositions d'objets et les objets individuels.

Pattern Etat (State)

Le pattern Etat permet à un objet de modifier son comportement quand son état interne change. Tout se passera comme si l'objet changeait de classe. Le pattern encapsule l'état dans des classes séparées et délègue à l'objet représentant l'état courant. Le pattern Etat est une solution qui permet d'éviter de placer une foule d'instructions conditionnelles dans notre contexte.

Pattern Proxy (Proxy)

Le pattern Proxy fournit un remplaçant à un autre objet, pour en contrôler l'accès. Il existe le Proxy **Distant**, le Proxy **Virtuel** et le Proxy de **Protection**. Le Proxy Distant contrôle l'accès à un objet distant. Le Proxy Virtuel contrôle l'accès à une ressource dont la création est coûteuse. Le Proxy de Protection contrôle l'accès à une ressource en fonction de droits d'accès.

Pattern Composé

Un pattern composé combine deux ou plusieurs patterns pour résoudre un problème général ou récurrent. MVC et Model 2 en font partie.

Pattern MVC ou Modèle-Vue-Contrôleur (Model-View-Controller) purement applicatif

Le pattern MVC sert à concevoir des interfaces graphiques modulaires et bien structurée. Il est composé de plusieurs patterns. Le Modèle utilise le pattern **Observateur** pour mettre à jour les Vues et les Contrôleurs. La Vue et le Contrôleur mettent en oeuvre le pattern **Stratégie**. Le Contrôleur est le comportement de la Vue, et il est facile de lui substituer un autre Contrôleur si l'on désire un comportement différent. La Vue elle-même emploie un pattern en interne pour gérer les fenêtres, les boutons et les autres composants de l'affichage : le pattern **Composite**.

Explication de l'utilisation des différents patterns dans MVC

Observateur

Le Modèle est Observable. Les Vues et le Contrôleur sont des Observateurs. En fait, tout objet intéressé par les changements d'état s'enregistre comme observateurs auprès du Modèle.

Stratégie

La Vue délègue au Contrôleur la gestion des actions de l'utilisateur. Le Contrôleur est la stratégie pour la Vue : c'est l'objet qui sait comment gérer les actions des utilisateurs. La Vue ne se soucie que de la présentation. Le Contrôleur se charge de traduire les entrées de l'utilisateur en actions et de les transmettre au Modèle.

Composite

La Vue est un composite constitué d'élements d'IHM (Interface Homme-Machine) (panneaux, boutons, champs de texte, etc...). Le composant de haut niveau contient d'autres composants qui contiennent eux-mêmes d'autres composants, et ainsi de suite jusqu'à ce qu'on atteigne les noeuds feuilles.

Adaptateur

Le pattern Adaptateur est utilisé dans MVC pour adapter un Modèle afin qu'il fonctionne avec des Vues et des Contrôleurs existants.

Pattern MVC Web ou Model 2

Le MVC Web est un peu différent de celui purement applicatif. MVC adapté au modèle navigateur/serveur est connue sous le nom “**Model 2**”. Model 2 sépare les composants comme dans le classique MVC mais il sépare aussi les responsabilités de production. Ceci permet d'éviter qu'un concepteur non aguéris à la programmation puisse modifier le code des servlets Java (alors qu'il n'y connais rien!). Les fichiers sont séparés. Le concepteur non programmeur n'a que à regarder les JSP simples qui contiennent de l'HTML et des JavaBeans sommaires. Le programmeur peut se concentrer sur les servlets.

Explication de l'utilisation des différents patterns dans MVC Web (Model 2)

Observateur

La Vue n'est plus un observateur au sens classique du terme ; autrement dit, elle ne s'enregistre pas auprès du Modèle pour être informée de ses changements d'état. En revanche, la Vue reçoit bien l'équivalent de notifications, mais elle les reçois indirectement, car c'est le contrôleur qui l'informe que le modèle à changé. Le contrôleur lui transmet même un bean qui lui permet d'extraire létat du modèle. Si on réfléchit au modèle du navigateur, on constate que la Vue n'a besoin d'être informée des changements d'état que lorsqu'une réponse HTTP est retournée au navigateur ; à tout autre moment, une notification serait inutile. Ce n'est que lorsqu'une page créée et retournée que la génération de la Vue et l'incorporation de l'état du modèle ont un sens.

Stratégie

Dans Model 2, l'objet Stratégie est toujours la servlet contrôleur, mais elle n'est pas composée directement avec la Vue de la façon classique. Cela dit, c'est un objet qui implémente un comportement pour la Vue et nous pouvons le remplacer par un autre contrôleur si nous voulons un comportement différent.

Composite

Comme dans notre IHM classique, la Vue est finalement constituée d'un ensemble de composants graphiques. En l'occurence, ils sont rendus par un navigateur Web à partir d'une description HTML, mais ce qui sous-tend le tout, c'est un système d'objets qui forment très probablement un composite.

Pattern Objet d'Accès aux Données (Data Access Object)

Les objets en mémoire vive sont souvent liés à des données persistantes (stockées en base de données, dans des fichiers, dans des annuaires, etc.). Le modèle DAO propose de regrouper les accès aux données persistantes dans des classes à part, plutôt que de les disperser. Il s'agit surtout de ne pas écrire ces accès dans les classes “métier”, qui ne seront modifiées que si les règles de gestion métier changent. Ce modèle complète le vieux modèle MVC (Modèle - Vue - Contrôleur), qui préconise

de séparer dans des classes différentes les problématiques : des “vues” (charte graphique, ergonomie) du “modèle” (cœur du métier) des “contrôleurs” (tout le reste : l'enchaînement des vues, les autorisations d'accès,... Avantages et inconvénients L'utilisation de **DAO** permet de s'abstraire de la façon dont les données sont stockées au niveau des objets métiers. Ainsi, le changement du mode de stockage ne remet pas en cause le reste de l'application. En effet, seules ces classes dites “techniques” seront à modifier (et donc à re-tester). Cette souplesse implique cependant un coût additionnel, dû à une plus grande complexité de mise en oeuvre.

Pattern Pont (Bridge)

Utilisez le **pattern Pont** pour faire varier non seulement vos implémentations, mais aussi vos abstractions. Le pattern Pont vous permet de faire varier l'implémentation et l'abstraction en plaçant les deux dans des hiérarchies de classes séparées.

Pattern Monteur (Builder)

Utilisez le **pattern Monteur** pour encapsuler la construction d'un produit et permettre de le construire par étapes. Vous souvenez-vous d'Itérateur ? Nous avions encapsulé l'itération dans un objet distinct et caché la représentation interne de la collection au client. L'idée est la même ici : nous encapsulons la création du planning dans un objet (appelons-le un monteur) et notre client demandera au monteur de construire la structure du planning à sa place.

Pattern Chaîne de Responsabilité (Chain of Responsability)

Utilisez le **pattern Chaîne de Responsabilité** quand vous voulez donner à plus d'un objet une chance de traiter une requête. Avec le pattern Chaîne de Responsabilité, vous créez une chaîne d'objets qui examinent une requête. Chaque objet considère la requête à son tour ou la transmet à l'objet suivant dans la chaîne.

Pattern Poids-Mouche (Flyweight)

Utilisez le **pattern Poids-Mouche** quand une instance d'une classe peut servir à fournir plusieurs « instance virtuelles ». Et si au lieu d'avoir des milliers d'objets Arbre vous pouviez reconcevoir votre système de façon à n'avoir qu'une seule instance d'Arbre et un objet client qui maintiendrait l'état de TOUS vos arbres ? Eh bien c'est le pattern Poids-Mouche.

Pattern Interprète(Interpreter)

Utilisez le **pattern Interprète** pour construire un interpréteur pour un langage. Quand vous devez implémenter un langage simple, le pattern Interprète permet de définir des classes pour représenter sa grammaire et un interpréteur pour interpréter les phrases. On utilise une classe pour représenter chaque règle du langage. Voici le langage Canard traduit en classes. Remarquez la correspondance

directe avec la grammaire.

Pattern Médiateur (Mediator)

Utilisez le **pattern Médiateur** pour centraliser le contrôle et les communications complexes entre objets apparentés. L'ajout d'un Médiateur au système permet de simplifier grandement tous les appareils : Ils informent le médiateur quand leur état change. Ils répondent aux requêtes du Médiateur.

Pattern Memento (Memento)

Utilisez le **pattern Memento** quand vous avez besoin de restaurer l'un des états précédents d'un objet, par exemple si l'utilisateur demande une « annulation ». Le Mémento a deux objectifs : Sauvegarder un état important d'un objet clé d'un système. Maintenir l'encapsulation de l'objet clé.

Pattern Prototype (Prototype)

Utilisez le **pattern Prototype** quand la création d'une instance d'une classe donnée est coûteuse ou compliquée. Le pattern Prototype permet de créer de nouvelles instances en copiant des instances existantes. (En Java, cela signifie généralement l'emploi de la méthode clone(), ou de la désérialisation quand on a besoin de copies en profondeur.) Un aspect essentiel de ce pattern est que le code client peut créer de nouvelles instances sans savoir quelle classe spécifique est instanciée.

Pattern Visiteur (Visitor)

Utilisez le **pattern Visiteur** quand vous voulez ajouter des capacités à un ensemble composite d'objets et que l'encapsulation n'est pas importante. Le Visiteur doit parcourir chaque élément du Composite : cette fonctionnalité se trouve dans un objet Navigateur. Le Visiteur est guidé par le Navigateur et recueille l'état de tous les objets du Composite. Une fois l'état recueilli, le Client peut demander au Visiteur d'exécuter différentes opérations sur celui-ci. Quand une nouvelle fonctionnalité est requise, seul le Visiteur doit être modifié.

Notes : pattern non étudié de manière complète via le livre

- Commande.
- Adaptateur.
- Façade.
- Patron de méthode.
- Itérateur.

- Composite.
- mvc applicatif (appli dj).

Boîte à outils

Bases de l'OO

- Abstraction.
- Encapsulation.
- Polymorphisme.
- Héritage.

Principes OO

- Encapsulez ce qui varie.
- Préférez la composition à l'héritage.
- Programmez des interfaces, non des implémentations.
- Efforcez-vous de coupler faiblement les objets qui interagissent (Faiblement couplé signifie que les objets sont indépendants les uns des autres. Les changements d'un objet n'obligent pas à faire tout un tas de changement dans d'autres objets).
- Les classes doivent être ouverte à l'extension mais fermées à la modification.
- Dépendez des abstractions. Ne dépendez pas des classes concrètes.
- Ne parlez qu'à vos amis : nous devons être attentif pour chaque objet au nombre de classes avec lesquelles il interagit et à la façon dont il entre en interaction avec elles.
- Ne nous appelez pas, nous vous appellerons : ce sont les super-classes qui décident et qu'elles ne doivent appeler les sous-classes que lorsqu'on en a besoin.
- Une classe ne doit avoir qu'une seule raison de changer. Limiter chaque classe à une seule responsabilité.