Conception Orientée Objet et Design Patterns - 1

4IRC
Version 2011/2012 - Release 2016-2017

F. PERRIN

membre de UNIVERSITÉ DE LYON

Préambule 4IRC

- Ce module est un module de la majeure Informatique et Systèmes.
- Il s'appuie sur le cours et les TPs du module POO du semestre 6 dont les concepts ont été consolidés à travers le projet de Génie Logiciel du semestre 7.
- Il est organisé en 8 séances de 4h alternant Cours/TP. Certains parties de cours seront à étudier en autonomie. Les TP sont essentiellement articulés autour d'un même projet.
- Un DS de 2h papier/crayon, des tests de connaissances permettront d'évaluer l'acquisition des compétences.
- Toutes les ressources sont disponibles sur le e-campus.

Remerciements

- Merci à Bruno MASCRET et Martine BREDA, pour leurs supports, conseils, exemples, schémas et relecture.
- Merci à Bruno MASCRET pour ses conseils en pédagogie active et ludique.
- Merci aux 4 auteurs de l'ouvrage *Design Patterns - Tête la première(O'Reilly edition)* dont je me suis inspirée.
- Merci à tous les photographes et artistes dont j'ai empruntés les œuvres.

I have a dream today

Ne serait-ce pas merveilleux si ce module me permettait de modéliser, concevoir, développer des programmes souples, extensibles, facile à maintenir sans me donner envie de déménager sur une île lointaine où les Design Patterns n'existent pas ? Mais ce n'est probablement qu'un rêve.

Quels bénéfices allez-vous tirer de ce module ?

- Distinguer les principaux patterns de conception.
- Apprendre à les utiliser : quand et comment les appliquer dans vos conceptions sans tomber dans la « patternite ».
- Arriver à les reconnaitre dans les API Java, les frameworks ou autres applications et comprendre enfin comment ils fonctionnent.

Comment allez-vous monter en compétence sans souffrir ?

- N'hésitez pas à rêver et à laisser parler vos émotions (surprise, curiosité, amusement, sensation de toute-puissance, etc.).
- Sollicitez activement vos neurones pour résoudre des problèmes, tirer des conclusions, etc.
- Voyez les exercices comme des défis et savourez la satisfaction de les avoir relevés.
- Plongez dans les TP pour capitaliser définitivement les compétences.

La bibliographie que je vous recommande

Sites Web d'approche facile

- http://conception.developpez.com/cours/
- http://rpouiller.developpez.com/tutoriel/java/design-patterns-gang-of-four
- http://www.dofactory.com/net/design-patterns/

Livres de référence :

- *[1] Design Patterns - Tête la première*, Eric Freeman , Elisabeth Freeman, Kathy Sierra et Bert Bates, 2004, O'Reilly Media.
- *[2] Design Patterns pour Java - les 23 modèles de conception*, Laurent Debrauwer, 2009, ENI.
- *[3] Design Patterns - Catalogue de modèles de conception réutilisables*, Erich Gamma , Richard Helm, Ralph Johnson et John Vlisside, 1999, Vuibert informatique.

COO : principes fondamentaux

Rappel : les piliers de l'approche Orientée Objet (1)

- Abstraction :
  - Que fait l'objet et quelles données il manipule ?
  - Sans se soucier des détails d'implémentation (stockage données, algorithme et langage pour traitement).
- Encapsulation :
  - Aspects externes accessibles à d'autres objets.
  - Implémentation interne invisible.
  - Evite interdépendances : modification implémentation n'affecte pas les applications qui emploient l'objet.

Rappel : les piliers de l'approche Orientée Objet (2)

- Héritage :
  - Mécanisme de généralisation/spécialisation permettant réutilisation. Le code est plus clair.
  - Classes dérivées héritent des données et fonctionnalités de la classe de base qu'elles pourront modifier et enrichir, sans la remettre en question.
- Polymorphisme :
  - Mécanisme permettant de manipuler des objets sans en connaitre (tout à fait) le type.
  - Permet à une même fonction d'avoir des comportements différents en fonction du type réel de l'objet manipulé.

Rappel : les piliers de l'approche Orientée Objet (3)

- Programmer une implémentation :
  - Chien c = new Chien ();
  - c.aboyer();
- Programmer une interface (au sens abstraction, « supertype ») :
  - Animal a = new Chien ();
  - a.parler ();
- Mieux encore, affecter l'objet de l'implémentation concrète au moment de l'exécution :
  - Animal a = fabrique.getAnimal(type); // new est effectué par l'objet // fabrique
  - a.parler(); // on sait que l'animal répond au message « parler() » // sans savoir de quel animal il s'agit

Pour une bonne conception (1)

- Pour que les programmes soient souples, extensibles et facile à maintenir, les classes doivent être :
  - Cohésives
  - Réutilisables
  - Compréhensibles
  - Maintenables
  - Indépendantes de la modification d'autres classes.

➡ Forte cohésion – Faible couplage

Pour une bonne conception (2)

- Cohésion :
  - Un composant, classe ou module, est fortement cohésif s'il est conçu autour d'un ensemble de fonctions apparentées ; il est faiblement cohésif si les fonctions n'ont pas de lien entre elles.
- Responsabilité unique :
  - Cohésion est principe plus général que Responsabilité Unique, mais les deux sont étroitement liés.
  - Les classes qui respectent ce principe sont plus facile à maintenir que celles qui assument plusieurs responsabilités

Pour une bonne conception (3)

- Couplage :
  - Le couplage exprime la relation étroite qu'un élément (Classe, système ou sous système) entretien avec un ou des autres éléments.
  - Un élément faiblement couplé ne possède pas ou peu de dépendances vis a vis d'autres éléments.
  - Un couplage trop fort entre deux éléments implique :
    - Modification de l'élément lié lors de la modification de l'autre.
    - Compréhension plus difficiles des éléments pris séparément,
    - Réutilisation contraignante du fait de la quantité d'éléments a intégrer pour en utiliser un seul.

Pour une bonne conception (4)

- Couplage fort :
  - Animal a;
    if (type == 1) a = new Chien;
    if (type == 2) a = new Chat;
  - // platsBrunch est un ArrayList
    for (int i = 0; i < platsBrunch.size(); i++) {
    Plat plat = (Plat)platsBrunch.get(i);
    }
- Couplage faible
  - Animal a;
    a = fabrique.getAnimal(type);
    L'objet fabrique gère les new.
  - // menuDejeuner est 1 Collection
    Iterator iterateur = menuDejeuner.iterator();
    while (iterateur.hasNext()) {
    Plat plat = (Plat)iterateur.next();
    }

Nous rencontrerons d'autres exemples au fil des cours.

Pour une bonne conception (5)

- Le pattern MVC (Modèle-Vue-Contrôleur) est un pattern d'architecture.
- Il est recommandé de l'utiliser pour développer tout système interactif manipulant :
  - Des données (Modèle).
  - La présentation de ces données (Vue).
  - La logique de gestion des événements entre les données et la présentation (Contrôleur).

- Objectif : pouvoir modifier une couche sans impact sur les autres (Forte cohésion – Faible couplage) :
  - Changement de l'IHM (Vue) sans impact sur les objets métiers (Modèle).
  - Des objets métiers (Modèle) réutilisables, sans modification, dans plusieurs applications (gérées par contrôleurs différents).

Principes fondamentaux de l'approche objet (1)

En vue d'une Forte cohésion et d'un Faible couplage, différents principes doivent être respectés, grâce notamment à la mise en place de Design patterns – Cf. chapitres suivants.

- Principe de responsabilité unique.
- Principe d'ouverture-fermeture :
  - Les classes doivent être ouvertes à l'extension mais fermées à la modification.
- Principe de substitution de Liskov :
  - Les fonctions qui utilisent des références vers des classes de base doivent pouvoir référencer des objets de classes dérivées sans les connaitre (polymorphisme).

Principes fondamentaux de l'approche objet (2)

- Principe de ségrégation des interfaces :
  - Toute classe implémentant une interface doit implémenter chacune de ses fonctions (abstraction et partition des responsabilités).

- Principe d'inversion de contrôle (injection de dépendances – illustré par principe d'Hollywood « Ne nous appelez pas, nous vous appellerons ») :
  - Permet de découpler les dépendances entre objets.
  - Les entités logicielles de haut niveau ne doivent pas dépendre des entités logicielles de bas niveau. Chacune doivent dépendre d'abstractions.

Quelques ERREURS de conception à éviter (1)

- L'instruction « switch » pour tester le type de l'objet :
  - ⇒ utiliser plutôt l'appel de fonctions polymorphes (animaux – parler()).
- Code en double :
  - ⇒ extraire le code commun dans une méthode unique.
  - ⇒ le plus souvent dans la super-classe (« est-un ») ou dans des classes implémentant des interfaces (« a un comportement »).

Quelques ERREURS de conception à éviter (2)

- Les objets « Dieu » :
  - 1 objet ayant de nombreuses responsabilités, les autres ne contenant que des données et accesseurs.
  - ⇒ redistribuer les responsabilités.
- Méthodes ou classes trop volumineuses :
  - ⇒ découper les responsabilités et définir de nouvelles classes et sous-classes.
- Méthode qui utilise plus de données venant de classes externes que de celle la contenant :
  - ⇒ déplacer la méthode.

Pré-requis de programmation nécessaires pour aborder chapitres suivants

- Récursivité.
- Classes, objets, constructeurs.
- Encapsulation et portée : public, protected, private.
- Classes dérivées, interfaces.
- Méthodes et classes abstraites. Méthode ne pouvant être redéfinies (final).
- Attributs et méthodes de classe (static).

Design Patterns :
Quoi ?
Pourquoi ?
Comment ?
Où ?
Combien ?

Pourquoi les Design Patterns ? (1)

Dialogue extrait de [1]

- « Développeur sceptique : Les patterns ne sont rien d'autre que l'application de bons principes OO…
- Gourou bienveillant : Erreur répandue, scarabée, mais c'est plus subtil encore. Tu as beaucoup à apprendre…
- Ds : O.K, hmm, pourquoi n'est-ce pas seulement une affaire de bonne conception objet ?

Je veux dire, tant que j'applique l'encapsulation et que je connais l'abstraction, l'héritage et le polymorphisme, est-ce que j'ai vraiment besoin des design patterns ?

Est-ce que ce n'est pas plus simple que cela ?

Est-ce que ce n'est pas la raison pour laquelle j'ai suivi tous ces cours sur l'OO ?

Je crois que les design patterns sont utiles pour ceux qui connaissent mal la conception OO.

Pourquoi les Design Patterns ? (2)

Dialogue extrait de [1]

- Gb : Ah, c'est encore un de ces malentendus du développement orienté objet : connaître les bases de l'OO nous rend automatiquement capables de construire des systèmes souples, réutilisables et faciles à maintenir.
- Ds : Non ?
- Gb : Non. En l'occurrence, la construction de systèmes OO possédant ces propriétés n'est pas toujours évidente, et seul beaucoup de travail a permis de la découvrir.
- Ds : Je crois que je commence à saisir. Ces façons de construire des systèmes orientés objets pas toujours évidentes ont été collectées…
- Gb : Oui, et constituent un ensemble de patterns nommés Design Patterns.
- Ds : Et si je connais les patterns, je peux me dispenser de tout ce travail et sauter directement à des conceptions qui fonctionnent tout le temps ?

Pourquoi les Design Patterns ? (3)

Dialogue extrait [1]

- Gb : Oui, jusqu'à un certain point. Mais n'oublie pas que la conception est un art. Un pattern aura toujours des avantages et des inconvénients. Mais si tu appliques des patterns bien conçus et qui ont fait leurs preuves au fil du temps, tu auras toujours beaucoup d'avance.
- Ds : Et si je ne trouve pas de pattern ?
- Gb : Les patterns sont sous-tendus par des principes orientés objet. Les connaître peut t'aider quand tu ne trouves pas de pattern qui corresponde à ton problème.
- Ds : Des principes ? Tu veux dire en dehors de l'abstraction, de l'encapsulation et…
- Gb : Oui, l'un des secrets de la création de systèmes OO faciles à maintenir consiste à réfléchir à la façon dont ils peuvent évoluer, et ces principes traitent de ces problèmes. »

Pourquoi les Design Patterns ? (4)

Connaître les bases de l'OO ne fait pas de vous un bon concepteur.
Les bonnes COO sont souples, extensibles et faciles à maintenir.
Les patterns vous montrent comment construire des systèmes OO de bonne qualité.

Pourquoi les Design Patterns ? (5)

- Les patterns résument une expérience éprouvée de la COO.
- Les patterns sont des solutions génériques aux problèmes de conception. Vous devez les adapter aux applications spécifiques.
- La plupart des patterns et des principes traitent des problèmes de changement dans le logiciel.
- La plupart des patterns permettent à une partie d'un système de varier indépendamment de toutes les autres. On essaie souvent d'extraire ce qui varie d'un système et de l'encapsuler.
- Les patterns fournissent un langage commun de communication avec les autres développeurs.

Les design patterns : c'est quoi ? (1)

- Un pattern est une solution à un problème dans un contexte :
  - Le contexte est la situation dans laquelle le pattern s'applique. Celle-ci doit être récurrente.
  - Le problème désigne le but que vous essayez d'atteindre dans ce contexte, ainsi que toutes les contraintes qui peuvent s'y présenter.
  - La solution est ce que vous recherchez : une conception générique que tout le monde peut appliquer et qui permet d'atteindre l'objectif en respectant l'ensemble des contraintes.

Les design patterns : c'est quoi ? (2)

- Exemple :
  - Le contexte : vous avez une collection d'objets.
  - Le problème : Vous devez parcourir ces objets sans exposer l'implémentation de la collection.
  - La solution : encapsulez l'itération dans une classe séparée.

Comment utiliser les patterns ?

- Laissez les Design Patterns émerger dans vos conceptions, ne les forcez pas pour le simple plaisir d'utiliser un pattern.
- Les Design Patterns ne sont pas gravés dans la pierre : adaptez-les selon vos besoins.
- Étudiez les catalogues de Design Patterns pour vous familiariser avec les patterns et les relations qui existent entre eux.
- ATTENTION :
  - L'abus de DP peut aboutir à un code exagérément élaboré.
  - Recherchez tjs la solution la plus simple et n'introduisez de patterns que lorsque le besoin s'en fait ressentir.

Les différents patterns existants (1) : historique

- Notion de patterns inventée par Christopher Alexander (Architecte - dans les années 70).
- Les patterns GRASP (General Responsibility Assignment Software Patterns (or Principles)) :
  - Décrivent règles pour affecter responsabilités aux classes pendant la conception.
  - En liaison avec la méthode de conception MVC « Modèle Vue Contrôleur »).
- Les patterns de conception formalisés dans le livre du « Gang of Four » (GoF, Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides) intitulé *Design Patterns – Elements of Reusable Object-Oriented Software* en 1995.

Les différents patterns existants (2) : les patterns GOF

- On distingue trois familles de patrons de conception :
  - Patterns créateurs.
  - Patterns structuraux.
  - Patterns comportementaux .
- Il existe 23 patrons de conception GOF :
  - Nous en étudierons certains en détail, d'autres à travers une étude bibliographique et des exercices.
  - Nous observerons comment le pattern d'architecture MVC est basé (a minima) sur les patterns Observer, Strategy et Composite.

Les différents patterns GOF (1)

- Patterns créateurs :
  - Ils définissent comment faire l'instanciation des objets et
  - Fournissent un moyen de découpler un client des objets qu'il a besoin d'instancier :
    - En le rendant indépendant de la façon dont les objets sont créés, composés, assemblés, représentés et
    - En cachant (encapsulant) ce qui est créé, qui crée, comment et quand.

Les différents patterns GOF (1)

- Patterns structuraux :
  - Ils définissent comment organiser (assembler) les classes d'un programme dans une structure plus large (séparant l'interface de l'implémentation).
  - Ces patterns sont complémentaires les uns des autres.
- Patterns comportementaux :
  - Ils définissent comment organiser les objets pour que ceux-ci collaborent (distribution des responsabilités) et expliquent le fonctionnement des algorithmes impliqués.

Les différents patterns existants (2) : liste des patterns GOF

- Patterns créateurs :
  - Abstract Factory.
  - Builder.
  - Factory Method.
  - Prototype.
  - Singleton.
- Patterns structuraux :
  - Adapter.
  - Bridge.
  - Composite.
  - Decorator.
  - Facade.
  - Flyweight.
  - Proxy.
- Patterns comportementaux :
  - Chain of Responsability.
  - Command.
  - Interpreter.
  - Iterator.
  - Mediator.
  - Memento.
  - Observer.
  - State.
  - Strategy.
  - Template Method.
  - Visitor.

Liste des patterns GOF triés par fréquence d'utilisation

- Patterns indispensables :
  - Abstract Factory.
  - Facade.
  - Factory Method.
  - Iterator.
  - Observer.
- Patterns très souvent utilisés :
  - Adapter.
  - Command.
  - Composite.
  - Proxy.
  - Singleton.
  - Strategy.
- Patterns souvent utilisés :
  - Bridge.
  - Decorator.
  - Prototype.
  - State.
  - Template Method.
- Patterns peu utilisés :
  - Builder.
  - Chain of Responsability.
  - Mediator.
- Patterns très peu utilisés :
  - Flyweight.
  - Interpreter.
  - Memento.
  - Visitor.

Comme la prose on les utilise sans le savoir ! Mais quand on en est conscient, on améliore sa réflexion…

« Par ma foi ! il y a plus de quarante ans que j'utilise des patterns sans que j'en susse rien, et je vous suis le plus obligé du monde de m'avoir appris cela. »

Les patterns de comportement

4IRC
Version 2011/2012 - Release 2016-2017

F. PERRIN

membre de UNIVERSITÉ DE LYON

Template Method

soit 1 algorithme qui a des parties invariables et des parties spécifiques aux types d'objet

Recette du café
(1) Faire bouillir de l'eau
(2) Filtrer le café à l'eau bouillante
(3) Verser le café dans une tasse
(4) Ajouter du lait et du sucre

Recette du thé
(1) Faire bouillir de l'eau
(2) Faire infuser le thé dans l'eau bouillante
(3) Verser le thé dans une tasse
(4) Ajouter du citron

Étudions l'exemple

Répertoire « patronMethode/baristasimple » [1]

- Cafe et The mènent la danse : ils contrôlent l'algorithme (la recette) .
- Le code est dupliqué entre Cafe et The.
- Changer l'algorithme nécessite d'ouvrir les sous-classes et d'apporter de multiples modifications.
- Les classes sont organisées selon une structure qui demande beaucoup de travail lorsqu'il faut ajouter une nouvelle boisson.
- La connaissance de l'algorithme et la façon de l'implémenter est répartie entre plusieurs classes.

Que faire des éléments communs ?

Quand nous avons du code dupliqué, c'est le signe qu'il faut réviser notre conception. Ici, il faut extraire les éléments communs pour les placer dans une classe de base puisque les recettes du café et du thé sont tellement similaires.

Que faire des éléments spécifiques (1) ?

- On constate qu'ils sont similaires et rentrent tous les 2 dans l'élaboration de la recette :
  - Préparer :
    - Passage du café ou
    - Infusion du thé.
  - AjouterSuppléments
    - Ajout du lait et du sucre ou
    - Ajout du citron.

Que faire des éléments spécifiques (2) ?

Identifiez les aspects de votre application qui varient et séparez-les de ceux qui demeurent constants.

Comment le DP Template method résout-il le Pb ?

- Le Patron de méthode fournit un cadre dans lequel on peut insérer d'autres boissons caféinées. Celles-ci n'auront qu'une paire de méthodes à implémenter.
- La classe BoissonCafeinee centralise la connaissance de l'algorithme et elle le protège. Elle s'en remet aux sous-classes pour fournir les implémentations complètes.
- L'algorithme réside à un seul endroit, et c'est uniquement là que les modifications du code ont lieu.

Étudions la solution (1)

Répertoire « patronMethode/barista [1] »

```
public abstract class BoissonCafeinee {
    final void suivreRecette() {          // Code protégé – ne peut être
        faireBouillirEau();               // redéfini dans classes dérivées
        preparer();                       // recette vraie
        verserDansTasse();                // ∀ la boisson
        ajouterSupplements();
    }
    abstract void preparer();             // Implémentée ds classes dérivées
    abstract void ajouterSupplements(); // Implémentée ds classes dérivées
    void faireBouillirEau() {
        System.out.println("Portage de l'eau à ébullition");
    }
    void verserDansTasse() {
        System.out.println("Remplissage de la tasse");
    }
}
```

Étudions la solution (2)

```
public class Cafe extends BoissonCafeinee {
    public void preparer() {
        System.out.println("Passage du café");
    }
    public void ajouterSupplements() {
        System.out.println("Ajout du lait et du sucre");
    }
}
public class The extends BoissonCafeinee {
    public void preparer() {
        System.out.println("Infusion du thé");
    }
    public void ajouterSupplements() {
        System.out.println("Ajout du citron");
    }
}
```


Pourquoi cela fonctionne-t-il ?

```
public class TestBoisson {
    public static void main(String[] args) {
      BoissonCafeinee the = new The();
      the.suivreRecette();
    }
}
```

- Le « client » (fonction qui utilise les objets – ici main()) instancie réellement un objet concret (celui qui est en bas de la hiérarchie d'objets – The ou Cafe).
- L'implémentation des méthodes nécessaires à suivreRecette() est alors connue (méthodes preparer() et ajouterSupplements() sont définies dans les classes dérivées concrètes).

Un peu de théorie
Pour chaque pattern on précise :

- Son nom et sa classe (création, structure, comportement).
- Son Intention : quelle est sa fonction (définition) ?
- Sa Motivation : description d'un pb et de la solution.
- Les Indications d'utilisation : Situations dans lesquelles le pattern peut être appliqué.
- Sa Structure : diagramme UML qui illustre les relations entre les classes qui participent au pattern.

Un peu de théorie
Pour chaque pattern on précise :

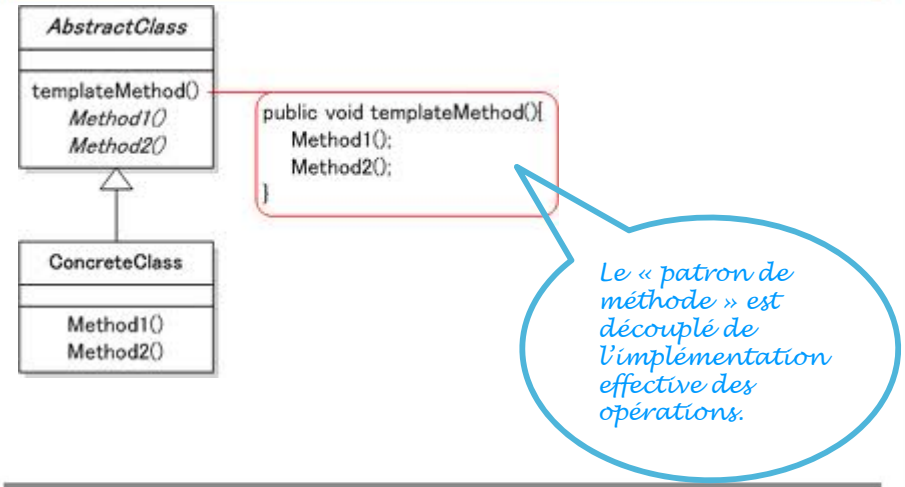
- Les participants : quelles classes et quelles sont leurs responsabilités.
- Les collaborations : comment les participants collaborent dans le pattern ?
- Les conséquences : effet positifs ou négatifs de son utilisation.
- Implémentation/Exemples de code.
- Les utilisations remarquables : exemples rencontrés dans les systèmes réels (API Java, etc.)
- Les patterns apparentés : relation qu'il entretient avec d'autres patterns.

DP Template Method : classification, intention, utilisation

- Classification : pattern comportemental.
- Intention :
  - Le pattern Template Method définit le squelette d'un algorithme dans une méthode, en déléguant certaines étapes aux sous-classes.
  - Ces étapes étant alors décrites dans les sous-classes sans modifier la structure de l'algorithme.
- Indication d'utilisation lorsque :
  - Une classe partage avec d'autres du code identique qui peut être factorisé.
  - Un algorithme possède une partie invariable et une partie spécifique à différents types d'objets.

DP Template Method : structure

http://www.itsenka.com/contents/development/designpattern/template_method.html



DP Template Method : participants, collaborations

- Participants :
  - La classe abstraite qui introduit la méthode « patron » et la signature des méthodes abstraites.
  - La ou les sous-classes concrètes implémentent les méthodes abstraites.
- Collaborations :
  - L'implémentation de l'algorithme est réalisé par la collaboration entre la méthode « patron » de la classe abstraite et les méthodes d'une sous classe concrète qui complètent l'algorithme.

DP Template Method : conséquences

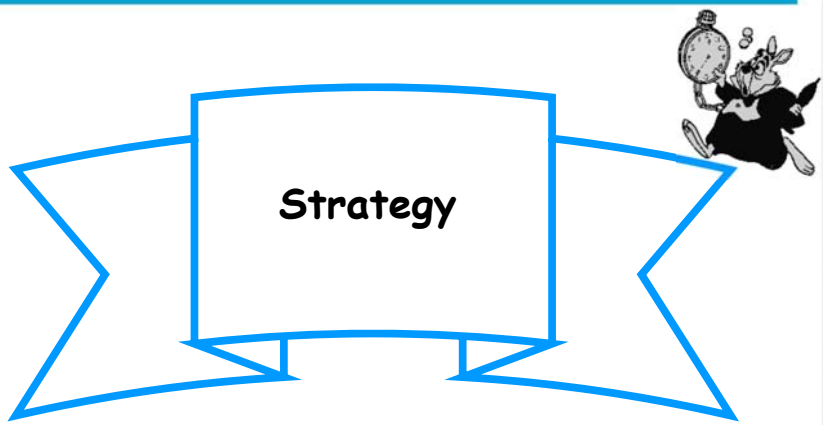
- Conséquences :
  - Les clients (utilisateur de ces objets) dépendront d'abstraction au lieu de dépendre de classe concrète, ce qui réduit les dépendances au niveau du système global.
  - La classe abstraite reste cependant dépendante de ces sous-classes (nécessité de surcharger les méthodes abstraites).

DP Template Method : utilisations remarquables, patterns connexes

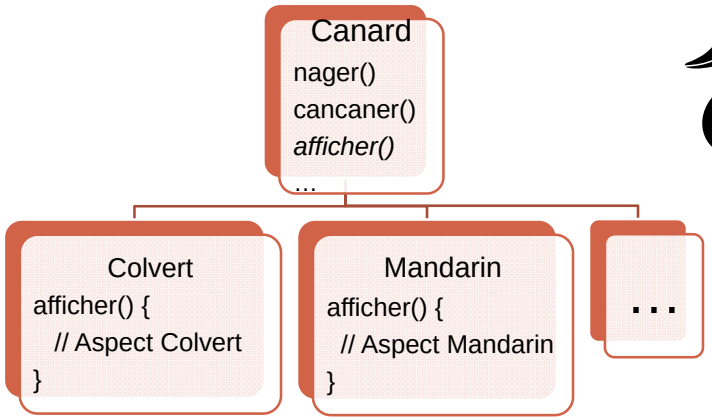
- Utilisations remarquables :
  - Toutes les méthodes non abstraites des classes AbstractList, AbstractSet, AbstractMap.
  - Méthode Arrays.sort() contient l'algorithme de tri et délègue à la méthode compareTo() des objets une partie de l'algo (ou à autre comparateur).
- Patterns connexes :
  - Strategy : délégation de l'algorithme complet.
  - Factory Method : les fabriques utilisent un patron de méthode en déléguant à leurs sous-classe la création des objets.

Principes objets illustrés par ce patterns

- Encapsulez ce qui varie pour que sa modification n'ait pas d'impact sur le reste.
- Respectez le principe d'Hollywood : « Ne nous appelez pas, nous vous appellerons » :
  - Éviter des dépendances circulaires explicites entre les composants de bas niveau et de haut niveau.
  - Permettre aux composants de bas niveau de s'adapter à un système où les composants de haut niveau déterminent quand on en a besoin et comment.



Pb : Soit un jeu de simulateur de marre au canard



Pb : Les concepteurs du jeu souhaitent de nombreuses autres fonctionnalités et types de canard

- Les canards doivent pouvoir voler.
  - ➡ Pas de Pb : on va ajouter une méthode voler() dans la classe Canard.
- Attention, il existe des canards en plastique, des leurres, etc. qui ne volent pas …
  - ➡ Pas de Pb : on va donner un comportement par défaut de voler() dans la classe de base et on le redéfinira dans les classes dérivées.
- Attention, du coup, tous les canards qui volent, volent peut être de la même façon, ou pas, et du code est dupliqué dans les classes dérivées…
  - ➡ L'héritage n'est, semble t-il, pas la bonne solution !

Que faire des éléments qui varient ? (1)

- Nous constatons que :
  - Le code serait dupliqué entre les sous-classes.
  - Les changements de comportement au moment de l'exécution semblent difficiles.
  - Il est difficile de connaître tous les comportements des canards.
  - Chaque modification du cahier des charges supposera des modifications à de nombreux endroits dans le hiérarchie de classe.
  - Si des Aigles ou des Moineaux doivent voler, il est dommage d'avoir caché le comportement voler() dans les Canards.

Que faire des éléments qui varient ? (2)

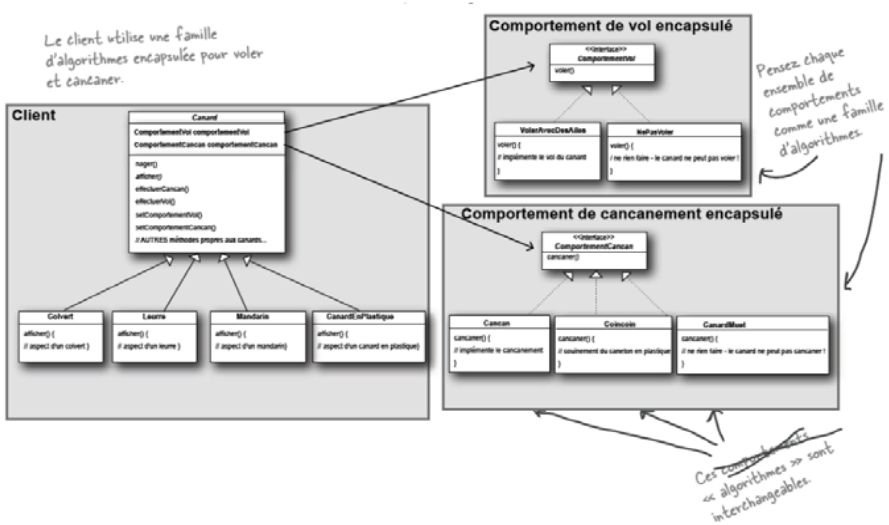
Extrayez ce qui varie et « encapsulez-le » pour ne pas affecter le reste de votre code.
Résultat ? Les modifications du code entraînent moins de conséquences inattendues et vos systèmes sont plus souples !

Comment le DP Strategy résout-il le Pb ? (1)

- Nous savons que voler() et cancaner() sont les parties de la classe Canard qui varient d'un canard à l'autre.
- Pour séparer ces comportements de la classe Canard, nous extrayons ces deux méthodes de la classe et nous créons un nouvel ensemble de classes pour représenter chaque comportement.
- Chaque ensemble de classes contiendra toutes les implémentations de leur comportement respectif. Par exemple, nous aurons :
  - Une classe qui implémente le cancanement,
  - Une autre le couinement,
  - Et une autre le silence.

Comment le DP Strategy résout-il le Pb ? (2)

Reproduction de [1]



Comment le DP Strategy résout-il le Pb ? (3)

- Désormais, les comportements de Canard résident dans une classe distincte — une classe qui implémente une interface comportementale particulière.

- Ainsi, les classes Canard n'ont besoin de connaître aucun détail de l'implémentation de leur propre comportement.
- La clé est qu'un Canard va maintenant déléguer ses comportements au lieu d'utiliser ceux qui étaient définies dans la classe Canard (ou une de ses sous-classes).

Comment le DP Strategy résout-il le Pb ? (4)

- Avec cette conception, les autres types d'objets peuvent réutiliser nos comportements de vol et de cancanement parce que ces comportements ne sont plus cachés dans nos classes Canard !
- Et nous pouvons ajouter de nouveaux comportements :
  - sans modifier aucune des classes comportementales existantes
  - ni toucher à aucune des classes Canard qui utilisent les comportements de vol.

On répond bien ainsi au principe d'Ouverture - Fermeture.

Étudions la solution (1)

Répertoire « strategie » [1]

```
public interface ComportementVol {
        public void voler();
}
public class VolerAvecDesAiles implements ComportementVol {
        public void voler() {
                System.out.println("Je vole !!");
        }
}
public interface ComportementCancan {
        public void cancaner();
}
public class Cancan implements ComportementCancan {
        public void cancaner() {
                System.out.println("Cancan");
        }
}
```


Étudions la solution (2)

```
public abstract class Canard {
        ComportementVol comportementVol;
        ComportementCancan comportementCancan;
        public Canard() {
        }
         public abstract void afficher();
        public void effectuerVol() {
                comportementVol.voler();
        }
        public void effectuerCancan() {
                comportementCancan.cancaner();
        }
        public void nager() {
                System.out.println("Tous les canards flottent, même les
                leurres!");
        }
}
```


Étudions la solution (3)

```
public class Colvert extends Canard {

        public Colvert() {

                comportementCancan = new Cancan();
                comportementVol = new VolerAvecDesAiles();
        }

        public void afficher() {
                System.out.println("Je suis un vrai colvert");
        }
}
```

Étudions la solution (4)

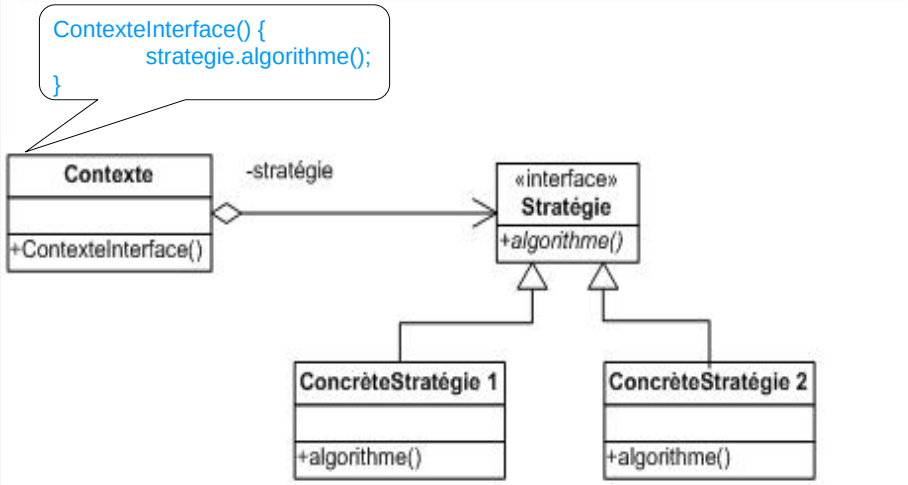
- La relation A-UN est une relation intéressante :
  - chaque canard a un ComportementVol et un ComportementCancan auxquels ils délègue le vol ou le cancanement.
- On utilise la composition au lieu de l'héritage.
  - Au lieu d'hériter leur comportement, les canards l'obtiennent en étant composés avec le bon objet comportemental.
- Créer des systèmes en utilisant la composition procure beaucoup plus de souplesse. Cela permet :
  - D'encapsuler une famille d'algorithmes dans leur propre ensemble de classes.
  - De modifier le comportement au moment de l'exécution tant que l'objet impliqué implémente la bonne interface comportementale.(à faire en exercice).

DP Strategy : classification, intention, indication

- Classification : pattern comportemental.
- Intention :
  - Le pattern Strategy définit une famille d'algorithmes, encapsule chacun d'eux et les rend interchangeables.
  - Strategy utilise la délégation pour décider quel algorithme choisir sans impact sur les interactions de l'objet avec ses clients.
- Indication d'utilisation lorsque :
  - Le comportement d'une classe peut être implémenté par différents algorithmes.
  - De nombreuses classes connexes ne diffèrent que par leur comportement.

DP Strategy : structure

- source http://blogs.codes-sources.com -



DP Strategy : participants

- Participants :
  - L'interface Strategy, commune à tous les algorithmes, est utilisée par le Contexte pour invoquer les algorithmes.
  - Ses sous-classes implémentent les algorithmes.
  - Contexte est la classe qui utilise un des algorithmes des classes d'implémentation de la Strategy. Elle possède une référence vers une instance d'une de ces classes.

DP Strategy : collaborations, conséquences

- Collaborations :
  - Le Contexte et les classes d'implémentation de Strategy interagissent pour implémenter les algorithmes.
  - Le client initialise le Contexte avec une instance de la classe d'implémentation de la Strategy, en général par passage de paramètre.
- Conséquences :
  - Les interactions entre l'objet et ses clients restent inchangées.
  - Risque d'accroissement du nb d'objets et des communication entre Stratégie et Contexte.

DP Strategy : utilisations remarquables, patterns connexes

- Utilisations remarquables :
  - Changement de comportement dans les jeux vidéo.
  - java.util.Comparator#compare(), exécuté par Collections#sort().
- Patterns connexes :
  - Template Method : il arrive que l'interface Strategy soit une classe abstraite utilisant le pattern Template Method.
  - State : tous deux impliquent la délégation à des comportements changeants.
  - Pattern architectural MVC : Strategy est utilisé pour implémenter le contrôleur.

Principes objets illustrés par ce patterns

- Préférez la composition à l'héritage.
- Programmer des interfaces (au sens abstraction – super-type) et non des implémentations.

State

Étudions l'exemple : soit un distributeur de bonbons

Répertoire « etat/bonbon » [1].

- L'état initial du distributeur est « Pas de pièce ».
- Si l'on insère une pièce, il passe à l'état « A une pièce ».
- Dans ce cas, si l'on tourne la poignée, il passe à l'état « Bonbon vendu », mais si l'on éjecte la pièce, il revient à l'état « Pas de pièce ».
- Il se peut que ce soit le dernier bonbon, auquel cas le distributeur passe à l'état « Plus de bonbon », sinon, il passe à l'état « Pas de pièce ».

- L'utilisateur pourrait aussi faire quelque chose d'absurde, par exemple essayer d'éjecter la pièce quand l'appareil est dans l'état « Pas de pièce » ou encore insérer deux pièces…

Quels défauts a cette approche ? (1)

- Nous constatons que :
  - Nous n'avons pas encapsulé ce qui varie.
  - Les ajouts ultérieurs sont susceptibles de provoquer des bogues dans le code.
  - Les transitions ne sont pas explicites. Elles sont enfouies au milieu d'un tas d'instructions conditionnelles.
  - Cette conception n'est pas orientée objet.
  - Ce code n'adhère pas au principe Ouvert-Fermé..

Comment le DP State résout-il le Pb ? (1)

- Il faut :
  - Isoler le comportement de chaque état dans sa propre classe.
  - Supprimer toutes les fâcheuses instructions conditionnelles qui auraient été difficiles à maintenir.
  - Fermer chaque état à la modification et laisser le Distributeur ouvert à l'extension en ajoutant de nouvelles classes.
  - Créer une base de code et une structure de classes qui sont plus faciles à lire et à comprendre.

Étudions la solution (1)

Répertoire « etat/bonbonetat » [1]

- Le pattern State est une solution qui permet d'éviter de placer une foule d'instructions conditionnelles dans un contexte.
- Avec le pattern State, nous sommes en présence d'un ensemble de comportements encapsulés dans des objets état.

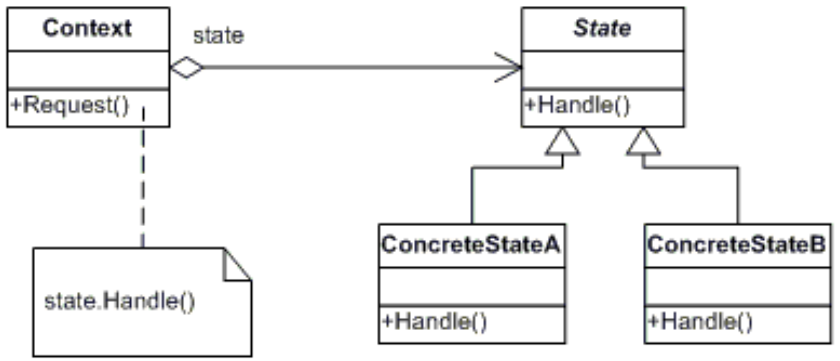
  À tout moment, le contexte délègue à l'un de ces états. Au fil du temps, l'état courant change pour l'un de ces objets état afin de refléter l'état interne du contexte, si bien que le comportement du contexte change également.
- Le client ne sait généralement pas grand-chose, voire rien, des objets état.
- En encapsulant les comportements dans des objets état, il suffit de modifier ces derniers dans le contexte pour changer son comportement.

DP State : classification, intention, utilisation

- Classification : pattern comportemental.
- Intention :
  - Encapsule des comportements basés sur des états et délègue le comportement à l'état courant.
- Indication d'utilisation lorsque :
  - Le comportement d'un objet dépend de son état qui change à l'exécution.
  - L'implémentation de cette dépendance à l'état par des tests est trop complexe.

DP State : structure

http://www.dofactory.com/Patterns/PatternState.aspx

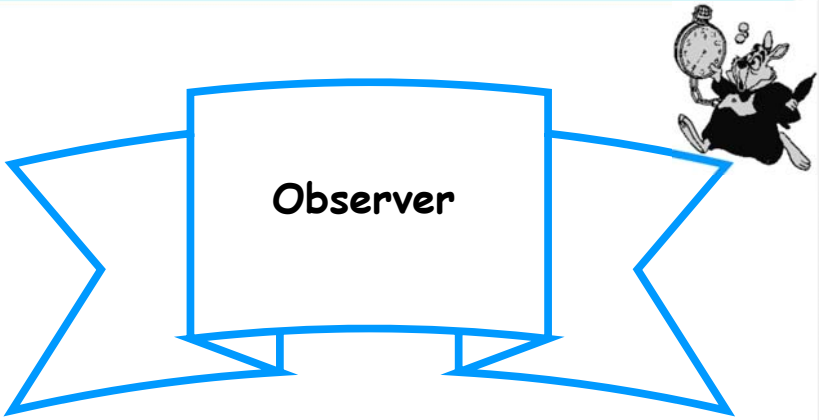


DP State : participants, collaborations

- Participants :
  - Contexte est une classe concrète (une machine à état) qui maintient une référence vers une instance d'une sous-classe de State qui définit l'état courant.
  - Etat est une classe abstraite qui introduit la signature des méthodes liées à l'état.
  - Ses sous-classes implémentent le comportement des méthodes relativement à chaque état.
- Collaborations :
  - La machine à états délègue les appels des méthodes dépendant de l'état courant vers un objet d'état.

DP State : conséquences, patterns connexes

- Conséquences :
  - Chaque état gère la transition vers l'état suivant.
  - Risque d'accroissement du nb d'objets et des communication entre State et Contexte.
- Utilisations remarquables :
  - Machines à états (distributeurs)
- Patterns connexes :
  - Strategy : tous deux impliquent la délégation à des comportements changeants.
  - Singleton : les états sont généralement des instances de Singleton auxquels on n'accède pas toujours via le pattern Singleton.



Pb : différents affichages de données météo à partir d'une unique source

- Plusieurs applications servent à afficher des données météorologiques qui intéressent un utilisateur ; parmi elles, le plug-in météo du navigateur internet et le widget qui s'affiche sur le bureau, etc.
- Un objet « données météo » se connecte de manière continue à une station distante afin de mettre à jour ses données. Il doit à chaque modification transmettre les nouvelles données aux applications d'affichage.
- Les données à afficher par le Widget sont la température, la pression et le taux d'humidité.
- Le plug-in quant à lui affiche la température et la prévision en fonction de l'évolution de la pression par rapport à celle précédemment relevée.

Pb : une implémentation de la classe « données météo » pourrait être la suivante

```
public class DonneesMeteo {
    // déclaration des attributs
    //…
    public void actualiserMesures() {
        temp = getTemperature();
        humidite = getHumidite();
        pression = getPression();
        widgetMeteo.actualiser(temp, humidite, pression);
        pluginMeteo.actualiser(temp, pression);
    }
    // autres méthodes de DonneesMeteo  : accesseurs et mutateurs
}
```

La classe « pousse » (push) ses données dans les classes WidgetMeteo et PluginMeteo

Quels défauts a cette 1ère approche ?

- Nous constatons que :
  - Nous n'avons pas encapsulé les parties qui varient.
  - Nous codons des implémentations concrètes, non des interfaces.
  - Nous devons modifier le code pour chaque nouvel élément d'affichage (ou suppression).
  - Les différents afficheurs n'ont pas une interface commune.

⇒ Pas de pb : nous allons faire le contraire : Ce sont les objets WidgetMeteo et PluginMeteo qui vont interroger l'objet DonneesMeteo dans leur méthode actualiser().

Quels défauts a cette 2ème approche ?

```
public class WidgetMeteo {
    // déclaration des attributs …
    public void actualiser() {
        temp = donneesMeteo.getTemperature();
        // …      }
    // autres méthodes de WidgetMeteo …
}
```

La classe « tire » (pull) ses données de la classe DonneesMeteo

- Problème :
  - A quelle fréquence la classe WidgetMeteo doit-elle interroger la classe DonneesMeteo ?
  - Quel que soit l'intervalle choisi :
    - soit les mesures ne seront pas précises.
    - soit on surchargera d'appels inutiles la classe DonneesMeteo.

Que faudrait-il faire pour résoudre ce Pb ?

Il faudrait trouver un moyen de coupler faiblement des objets afin que lorsque l'un change d'état, tout ceux qui en dépendent en soient notifié et soient mis à jour automatiquement.

Comment le DP Observer résout-il le Pb ? (1)

- Le DP Observer est basé sur la métaphore publisher/subscriber. Ex :
  - Un éditeur se lance dans les affaires et commence à diffuser des journaux.
  - Vous souscrivez un abonnement. Chaque fois qu'il y a une nouvelle édition, vous la recevez. Tant que vous êtes abonné, vous recevez de nouveaux journaux.
  - Quand vous ne voulez plus de journaux, vous résiliez votre abonnement. On cesse alors de vous les livrer.
  - Tant que l'éditeur reste en activité, les particuliers, les hôtels, les compagnies aériennes, etc., ne cessent de s'abonner et de se désabonner.

Comment le DP Observer résout-il le Pb ? (2)

- Dans l'exemple précédent, le Sujet (l'observable) est l'éditeur et les abonnés les Observateurs.
- L'objet Sujet gère une donnée quelconque.
- Les observateurs ont souscrit un abonnement (se sont enregistrés) auprès du Sujet pour recevoir les mises à jour quand les données du Sujet changent.
- Quand les données de l'objet Sujet changent, les observateurs en sont informés et les nouvelles valeurs sont communiquées aux observateurs (souvent par « un pull »).

Étudions la solution (1)

Répertoire « Meteo» [1]

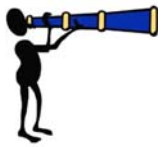
```
public abstract class Sujet {
    protected List<Observateur> observateurs = new
                                        ArrayList<Observateur>();
    public void ajoute(Observateur observateur) {
            observateurs.add(observateur);
    }
    public void retire(Observateur observateur) {
            observateurs.remove(observateur);
    }
    public void notifie() {
            for (Observateur observateur : observateurs)
                observateur.actualise();
    }
}
```


Étudions la solution (2)

```
public class DonneesMeteo extends Sujet {
    protected double temperature;          // ce qui est observé
     …
    public double getTemperature() {
        return this.temperature;
    }
    public void setTemperature(double temperature) {
        this. temperature = temperature;
        this.notifie();     // informe les observateurs d'un changement d'état
    }
    …
}
```


Étudions la solution (3)

```
public interface Observateur {
        void actualise();
}
```



Étudions la solution (4)

```
public class WidgetMeteo implements Observateur  {
    protected DonneesMeteo donneesMeteo;
    protected double temperature;
    public WidgetMeteo (DonneesMeteo donneesMeteo) {
            this.donneesMeteo = donneesMeteo;
            this.donneesMeteo.ajoute(this);
            this.actualise();
    }
    public void actualise()  {
            temperature = donneesMeteo.getTemperature();
            affiche();
    }
    public void affiche()  {  … }
}
```

Étudions la solution (5)

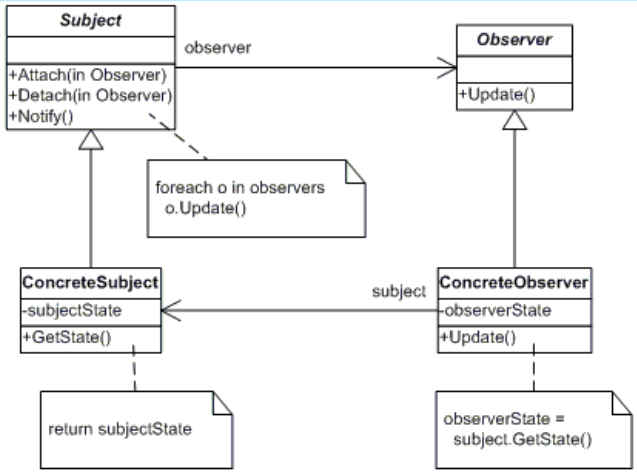
```
public class Utilisateur {
    public static void main(String[] args) {

        DonneesMeteo donneesMeteo = new DonneesMeteo();
        donneesMeteo.setTemperature(24.00);
        WidgetMeteo w = new WidgetMeteo(donneesMeteo);

        // si pas effectué dans la classe de l'observer
        // donneesMeteo.ajoute(w);

        donneesMeteo.setTemperature(25.00); // invoque w.actualise()
    }
}
```


DP Observer : classification, intention, utilisation

- Classification : pattern comportemental.
- Intention :
  - Le DP Observer construit une dépendance entre un Sujet et des Observateurs de sorte que chaque modification du Sujet soit notifiée aux Observateurs afin qu'il soient modifiés automatiquement.
- Indication d'utilisation lorsque :
  - Une modification dans l'état d'un objet engendre des modifications dans d'autres objets.
  - Un objet veut prévenir d'autres objets sans devoir connaitre leur type, i.e. sans être fortement couplés avec eux.

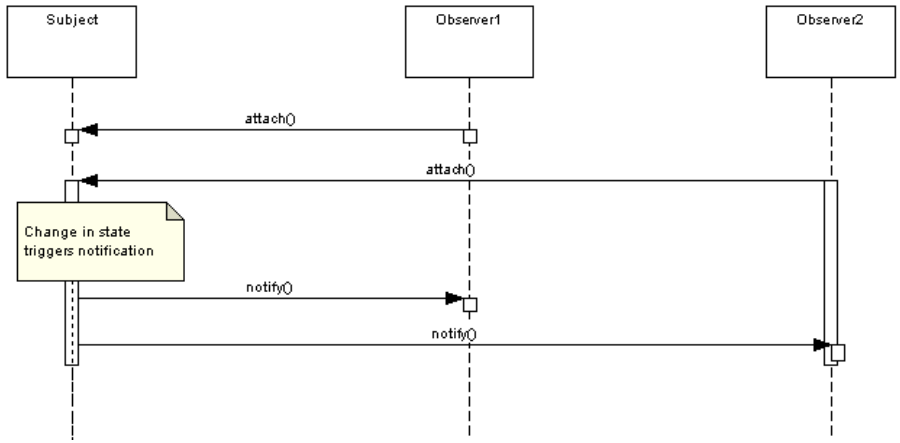
DP Observer : structure

http://www.dofactory.com/Patterns/PatternObserver.aspx



DP Observer : diagramme de séquence

https://dzone.com/articles/design-patterns-uncovered



DP Observer : participants

- Participants :
  - Le Sujet est la classe abstraite (interface) qui introduit l'association avec les observateurs ainsi que les méthodes pour ajouter et retirer des observateurs.
  - L'Observateur est l'interface à implémenter pour recevoir les notifications (update()).
  - Le SujetConcret est une implémentation d'un Sujet. Il envoie une notification quand son état est modifié.
  - L'ObservateurConcret est une implémentation de l'Observateur. Il maintient une référence vers son SujetConcret et implémente la méthode update().

DP Observer : collaborations, conséquences

- Collaborations :
  - Le SujetConcret notifie ses observateurs lorsque son état est modifié.
  - Lorsqu'un ObservateurConcret reçoit cette notification, il se met à jour en conséquence.
- Conséquences :
  - Les Observateurs sont faiblement couplés, au sens où l'Observable (le Sujet) ne sait rien d'eux, en dehors du fait qu'ils implémentent l'interface Observer.

Le pouvoir du Couplage faible (1)

- Les conceptions faiblement couplées permettent :
  - de construire des systèmes OO souples,
  - capables de faire face aux changements parce qu'ils minimisent l'interdépendance entre les objets.
- Le pattern Observateur permet une conception dans laquelle le couplage entre sujets et observateurs est faible. Pourquoi ?
  1. Le sujet ne sait qu'une chose à propos de l'observateur :
     - Il implémente l'interface Observateur.
     - Il n'a pas besoin de connaître, ni la classe concrète de l'observateur, ni ce qu'il fait ni quoi que ce soit d'autre.

Le pouvoir du Couplage faible (2)

2. Comme le sujet dépend uniquement d'une liste d'objets qui implémentent l'interface Observateur, nous pouvons ajouter/supprimer des observateurs à volonté.
3. Nous n'avons jamais besoin de modifier le sujet pour ajouter de nouveaux types d'observateurs.
   - Il suffit d'implémenter l'interface Observateur dans la nouvelle classe et de l'enregistrer en tant qu'observateur.
   - Le sujet ne s'en soucie aucunement : il continuera à diffuser des notifications à tous les objets qui implémentent l'interface Observateur.
4. Les modifications des sujets n'affectent pas les observateurs et inversement.

Le pouvoir du Couplage faible (3)

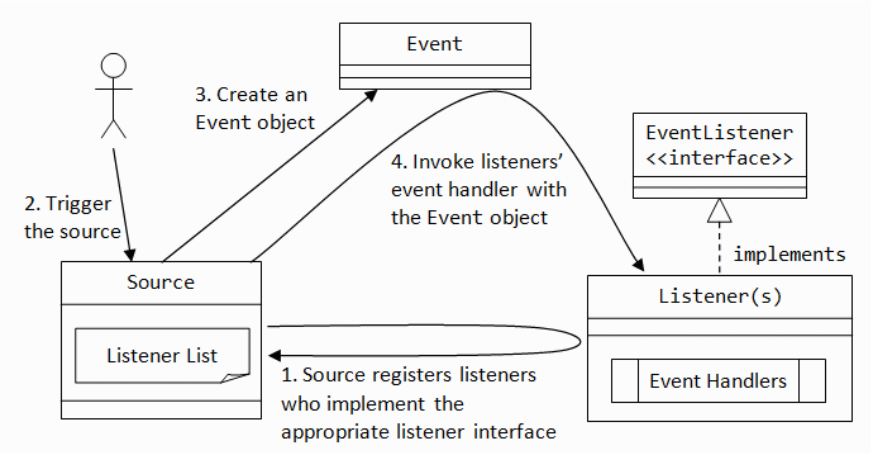
- Cependant, le pattern Observateur introduit un couplage plutôt fort entre l'observateur concret et le sujet concret.
  - Il faut éviter autant que possible que l'observateur concret ait une référence vers le sujet concret, sinon, il peut invoquer n'importe quelle méthode sur le sujet concret à son insu. Préférer, a minima, une référence vers un Sujet (Observable abstrait).
  - Il faut que le sujet concret envoie directement, lors de la notification à ses observateur, l'information désirée (éventuellement sous une forme épurée).

DP Observer : utilisations remarquables, patterns connexes

- Utilisations remarquables :
  - Pour détecter un événements qui interviennent sur un composant Swing, java utilise DP Observer.
  - Dans les JavaBeans.
- Patterns connexes :
  - Mediator : encapsule également les mises à jour complexes.
  - Pattern architectural MVC : le Modèle est observé par la Vue.

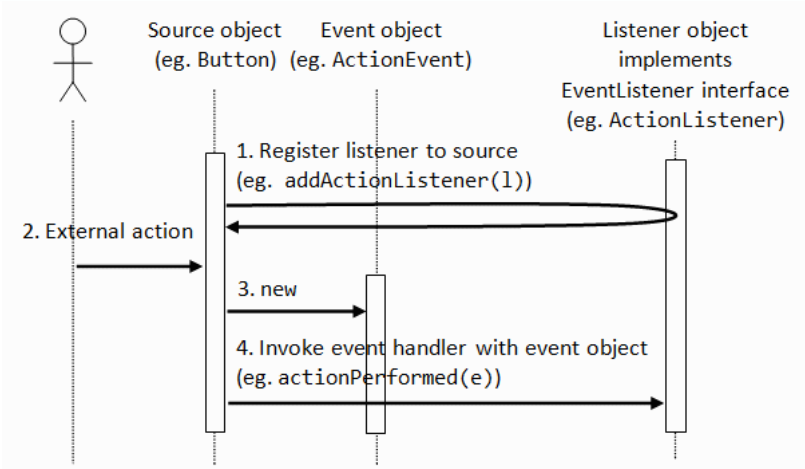
DP Observer : mise en œuvre dans Java (1)

http://www3.ntu.edu.sg/home/ehchua/programming/java/J4a_GUI.html



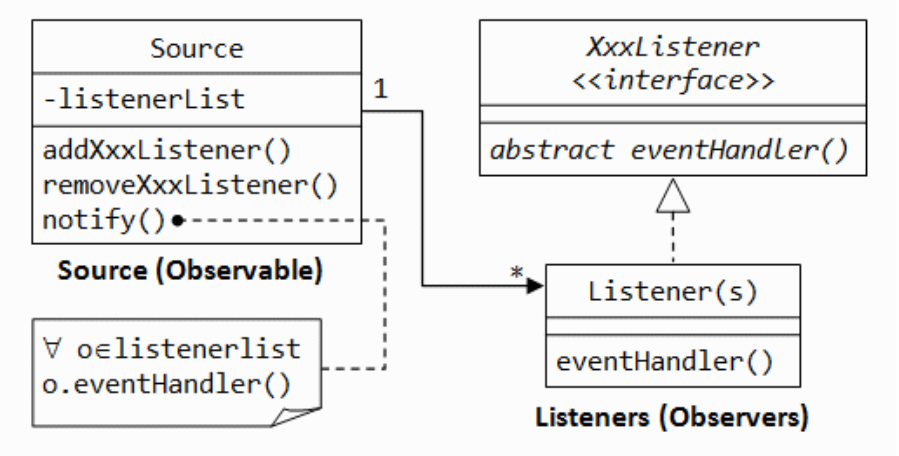
DP Observer : mise en œuvre dans Java (2)

http://www3.ntu.edu.sg/home/ehchua/programming/java/J4a_GUI.html



DP Observer : mise en œuvre dans Java (3)

http://www3.ntu.edu.sg/home/ehchua/programming/java/J4a_GUI.html

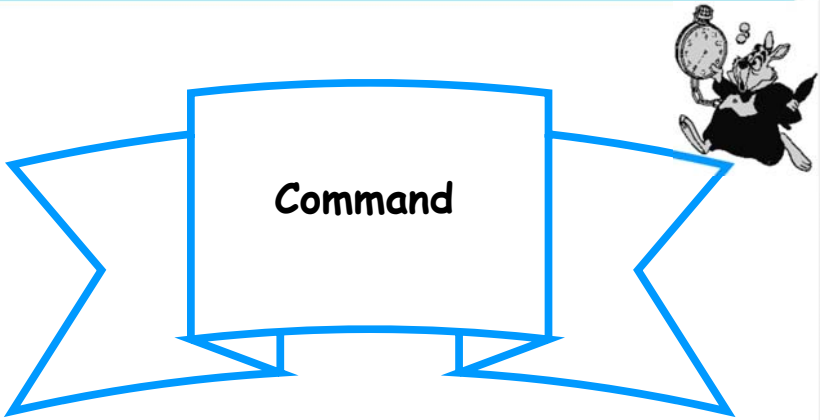


DP Observer : pour aller plus loin

- Il existe l'interface Observer et la classe Observable du package java.util. Peuvent être utile, mais :
  - Observable est une classe (pas une interface) qui doit donc être dérivée. Or une classe ne peut pas hériter de 2 classes en java ce qui en limite l'usage.
  - Elle peut cependant être utilisée comme une Strategy : ça complexifie un peu le code mais permet de s'intégrer dans une hiérarchie d'héritage.
  - Elle envoie une référence de l'objet observé aux observateurs mais permet aussi d'envoyer un objet d'échange d'information.

Principes objets illustrés par ce patterns

Efforcez-vous de coupler faiblement les objets qui interagissent.



DP Command encapsule l'invocation des méthodes (1)

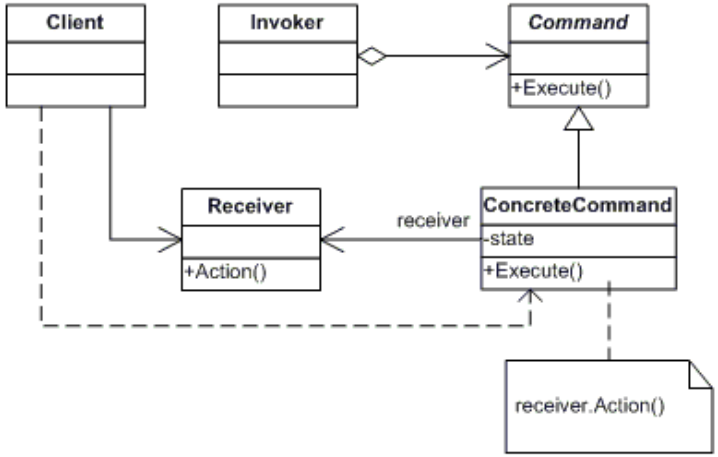
- Le pattern Commande découple un objet émettant une requête de celui qui sait comment l'exécuter.
- Un objet Commande est au centre du mécanisme de découplage et encapsule un récepteur avec une action (ou un ensemble d'actions).
- Un invocateur requiert un objet Commande en appelant sa méthode executer(), laquelle invoque ces actions sur le récepteur.
- Les invocateurs peuvent être paramétrés par des Commandes, même dynamiquement lors de l'exécution.

DP Command encapsule l'invocation des méthodes (2)

- Les Commandes peuvent prendre en charge l'annulation en implémentant une méthode annuler() qui restitue à l'objet l'état qui était le sien avant le dernier appel de la méthode executer().
- Les MacroCommandes sont une simple extension de Commande qui permet d'invoquer plusieurs commandes. Les macrocommandes supportent également l'annulation.
- Dans la pratique, il n'est pas rare que des objets Commande « intelligents » implémentent eux-mêmes le traitement de la requête au lieu de la déléguer à un récepteur.

DP Command : structure

http://www.dofactory.com/Patterns/PatternCommand.aspx



DP Command : utilisations remarquables

- Utilisations remarquables :
  - On peut utiliser des Commandes pour implémenter des mécanismes de journalisation ou des systèmes transactionnels.
  - Dans toutes les implémentations de java.lang.Runnable.

DP Command

Étude en autonomie du cours sur le pattern Command

1ers pas :

http://www.developpez.net/forums/d666748/general-developpement/alm/design-patterns/introduction-simple-douceur/

Pour le TP :

http://zenika.developpez.com/tutoriels/java/patterns-command/

Mediator

DP Mediator

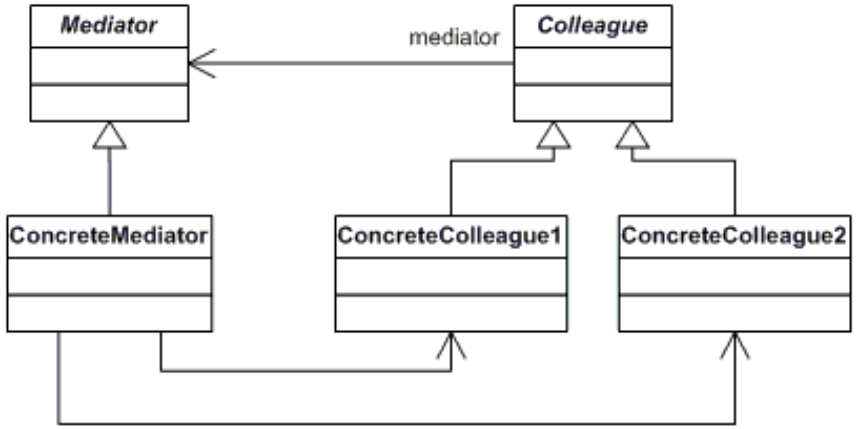
- Utilisation :
  - Utilisez le pattern Médiateur pour centraliser le contrôle et les communications complexes entre objets apparentés.
  - Médiateur est couramment employé pour coordonner des composants d'IHM.
- Inconvénients :
  - En l'absence de conception soigneuse, l'objet Médiateur lui-même peut devenir exagérément complexe.

DP Mediator

- Avantages :
  - Augmente la réutilisabilité des objets pris en charge par le Médiateur en les découplant du système.
  - Simplifie la maintenance du système en centralisant la logique de contrôle.
  - Simplifie et réduit la variété des messages échangés par les différents objets du système.

DP Mediator : structure

http://www.dofactory.com/Patterns/PatternMediator.aspx



Iterator

Pb : différents moyens de ranger/stocker les éléments dans les agrégats

- Il y a des quantités de façon de placer des objets dans un agrégat : tableaux (au sens vecteur), piles, listes, tables de hachage. Chacune a ses avantages et ses inconvénients.
- De même, il y a 2 façons de stocker les données en mémoire : tableau ou liste chainée.
- Mais il y aura toujours un moment où votre client voudra opérer des itérations sur ces objets. Allez-vous alors lui montrer votre implémentation ? Espérons que non ! Ce serait un manque de professionnalisme absolu…

Que faudrait-il faire pour résoudre ce Pb ?

Problème :
Comment fournir un mécanisme permettant de parcourir une séquence sans en exposer l'organisation ni la représentation ?

Solution :
Encapsulez l'itération et déchargez l'agrégat de la responsabilité de la navigation dans ses données.

Comment le DP Iterator résout-il le Pb ? (1)

- Le DP Iterator fournit un moyen d'accéder aux éléments d'un agrégat sans exposer sa structure interne (mode de rangement et/ou mode de stockage).
- Un Itérateur a pour tâche de parcourir un agrégat et encapsule l'itération dans un autre objet. L'agrégat en est donc déchargé.
- Un Itérateur fournit une interface commune pour parcourir les éléments d'un agrégat. Il permet ainsi d'exploiter le polymorphisme pour écrire le code qui utilise ces éléments.

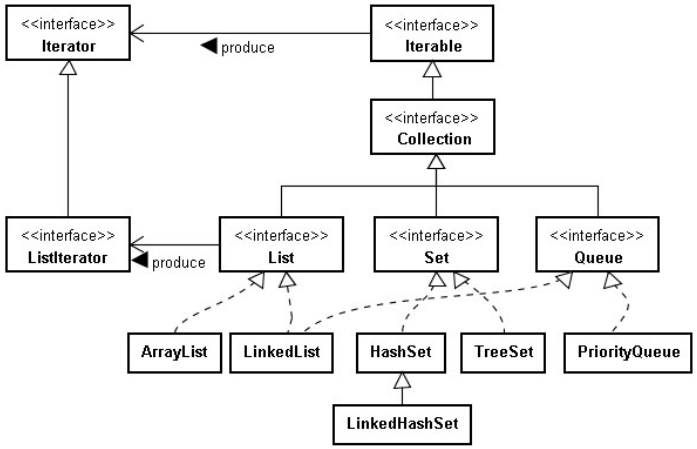
Étudions la solution (1)

Répertoire « Iterator »

- Java met en œuvre le DP Iterator :

  - Pour être parcouru par un Itérateur, un agrégat doit implémenter l'interface Iterable. Cette dernière définit une unique méthode : Iterator<T> iterator();
  - Toute classe qui implémente cette interface doit donc en redéfinir la méthode iterator(). Celle-ci retourne un objet implémentant l'interface Iterator.
  - L'interface Iterator spécifie le comportement d'un iterateur unidirectionnel, l'interface ListIterator celui d'un iterateur bidirectionnel dans des listes doublement chainées.
  - Dans chaque classe devant être parcourue, il suffit de définir une classe interne (éventuellement anonyme) qui implémente l'interface Iterator (ou ListIterator).

Hiérarchie des classes du Framework de collections Java



Étudions la solution (2)

Interface Iterator :

```
Public interface Iterator {
    public boolean    hasNext() ;
    public object     next() ;  // retourne l'élément et passe au suivant
    public void       remove() ;
}
```

- Rq : vous pouvez définir votre propre interface d'Itérateur, ajouter une méthode creerIterator() dans vos classes, etc.

Étudions la solution (3)

- Interface ListIterator // bidirectionnel

```
Public interface ListIterator extends Iterator {
    public void      add(Object object);
    public boolean   hasNext() ;
    public boolean   hasPrevious();
    public object    next() ; // retourne l'élément et passe au suivant
    public int       nextIndex();
    public Object    previous(); // passe au précédent et retourne elt
    public int       previousIndex();
    public void      remove() ;
    public void      set(Object object);
}
```


Étudions la solution (4)

Soit vecteur définit par : Collection collection = new ArrayList();

- Parcours en utilisant les méthodes de la classe :

```
for (int i = 0; i < collection.size(); i++)
  {     System.out.println(collection.get(i).toString());   }
```

- Utilisation d'un itérateur :

```
for (Iterator it = collection.iterator(); it.hasNext(); )
  {     System.out.println(it.next().toString());   }
```

- La boucle for-each, permet d'utiliser une boucle *for* spécifique pour itérer n'importe quelle classe java qui implémente l'interface Iterable.

```
for(Object valeur : collection)
  {     System.out.println(valeur.toString());   }
```


Étudions la solution (5)

- Implémentation d'un itérateur par une classe interne nommée :

```
Public class MaClasse implement Iterable {
    /* …. attributs et méthodes de la classe … */
    public Iterator iterator() {
            return new MonIterator() ;
    }
    private class MonIterator  {    // classe qui implémente Iterator
            public boolean hasnext() {  // ...;  }
            public Object next() {  // ...;  }
            public void remove() { throw new
                    UnsupportedOperationexception();}
    } ;
}
```


Étudions la solution (6)

- Implémentation d'un itérateur par une classe anonyme :

```
Public class MaClasse implement Iterable {
  /* …. attributs et méthodes de la classe … */
  public Iterator iterator() {
      return new Iterator() {   // classe anonyme qui implémente
                                // l'interface Iterator
          public boolean hasnext() {  // ...;  }
          public Object next() {  // ...;  }
          public void remove() {throw new
                  UnsupportedOperationexception();}
    } ;
  }
}
```

Étudions la solution (7)

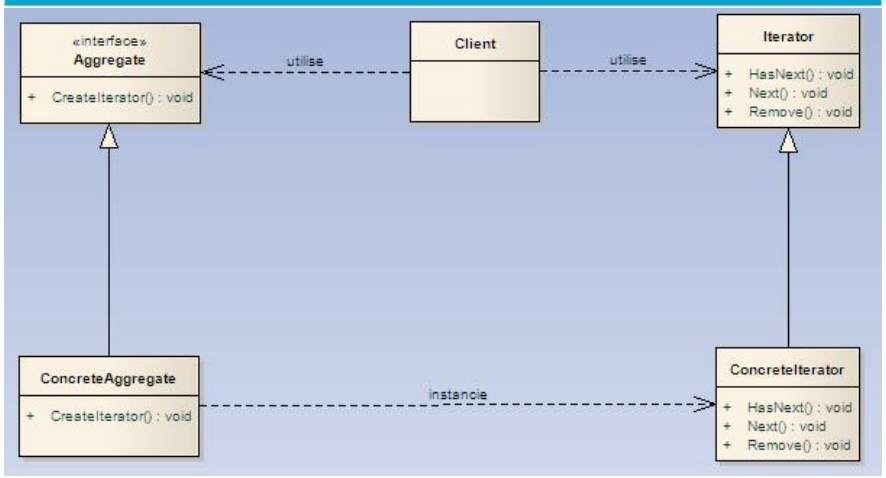
- Etude d'un pgm de résolution du Pb de Josephus qui utilise une liste circulaire bidirectionnelle :
  - Des soldats juifs, cernés par des soldats romains, décident de former un cercle. Un premier soldat est choisi au hasard et est exécuté, le troisième à partir de sa gauche (ou droite) est ensuite exécuté. Tant qu'il y a des soldats, la sélection continue.
  - Le but est de trouver à quel endroit doit se tenir un soldat pour être le dernier. Josèphe, peu enthousiaste à l'idée de mourir, parvint à trouver l'endroit où se tenir.

DP Iterator : classification, intention, utilisation

- Classification : pattern comportemental.
- Intention :
  - Le DP Iterator fournit à ses clients le moyen d'accéder de manière séquentielle aux éléments d'un agrégat d'objets sans exposer sa structure interne.
- Indication d'utilisation lorsque :
  - Le parcours d'accès au contenu d'une collection doit être réalisé sans accéder à la représentation interne de la collection.
  - Il doit être possible de gérer plusieurs parcours simultanément sur la même collection.

Structure du DP Iterator

http://blog.developpez.com/youssef/p9241/design-pattern/collectional-patterns/iterator-pattern



DP Iterator : participants

- Participants :
  - Iterator est l'interface qui introduit la signature des méthodes qui permettent de parcourir les éléments d'une collection.
  - ConcreteIterator doit implémenter les méthodes de l'interface Iterator. Il est responsable de la gestion de la position courante de l'itération
  - Agregate est la classe abstraite qui implémente l'association de la collection d'éléments et la méthode de création d'un Iterator (iterator()).
  - ConcreteAgreagate est la collection instanciée qui définit comportement de l'Iterator ds 1 classe interne.

DP Iterator : collaborations, conséquences

- Collaborations :
  - L'Itérateur garde en mémoire l'objet courant dans la collection.
  - Il est capable de « calculer » l'objet suivant du parcours.
- Conséquences :
  - La représentation sous-jacente de la collection peut changer sans que cela n'affecte les clients puisque la collection est découplée de son implémentation.

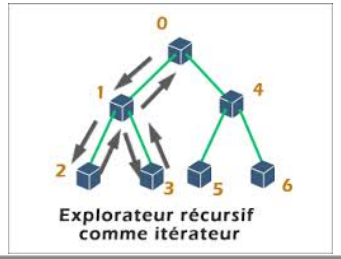
DP Iterator : utilisations remarquables

- Utilisations remarquables :
  - Utilisation dans le framework de Collections Java :
    - public interface Iterable<T> dans java.lang et public interface Iterator<E> dans java.util.
    - Les méthodes des collections font appel aux itérateurs pour leur propre fonctionnement :

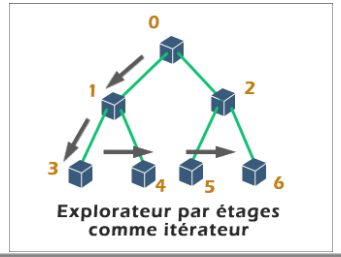
```
public int hashCode() {            // ds n'importe quelle classe
    int n = 0;
    for (Iterator it = this.iterator(); it.hasNext(); ) {
        Object object = it.next();
        if (object != null) n += object.hashCode();
    }
    return n;
}
```


DP Iterator : patterns connexes

- Patterns connexes :
  - Composite : des Itérateurs peuvent être fournis pour traverser une structure Composite.
  - Abstract Factory : structure similaire : c'est une sous classe qui décide de l'objet à créer.



Explorateur récursif comme itérateur



Explorateur par étages comme itérateur

Principes objets illustrés par ce patterns

Rappel :

Principe de Responsabilité Unique :
Une classe ne doit avoir qu' 1 seule raison de changer.

Visitor

Que faire des éléments qui varient ? (2)

Problème :

Comment ajouter des fonctionnalités à un Composite d'objets ?

Solution :

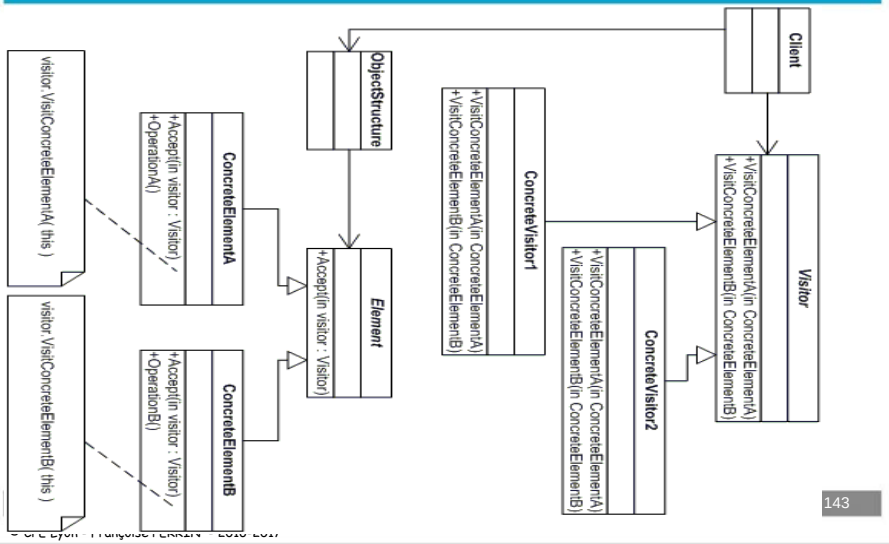
Déléguez le comportement à une classe externe (un Visiteur) et autorisez la à visiter les composants.

Comment le DP Visitor résout-il le Pb ?

- Le client demande au Visiteur des opérations sur le Composite.
- Le visiteur va demander aux Composants de lui retourner leur état et s'occupe du traitement.
- Les composants doivent :
  - Implémenter une fonction qui retourne cet état.
  - Accepter un visiteur.
- Le code des opérations exécutées par le visiteur est ainsi centralisé et la structure du Composite n'est pas modifiée.

DP Visitor : structure

http://www.dofactory.com/Patterns/PatternVisitor.aspx



DP Visitor

Étude en autonomie du cours du Site developpez.com sur

le pattern Visitor

http://rpouiller.developpez.com/tutoriel/java/design-patterns-gang-of-four

Chain of Responsibility

DP Chain of Responsability

- Principe :
  - 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.

DP Chain of Responsability

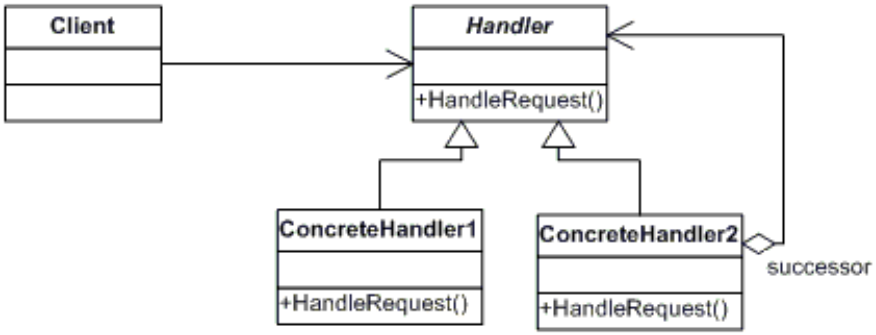
- Avantages :
  - Découple l'émetteur de la requête de ses récepteurs.
  - Simplifie votre objet, car il n'a pas besoin de connaître la structure de la chaîne ni de conserver des références directes à ses membres.
  - Permet d'ajouter ou de supprimer dynamiquement des responsabilités en changeant les membres ou l'ordre de la chaîne.

DP Chain of Responsability

- Emploi :
  - Couramment utilisé dans les interfaces graphiques pour gérer des événements comme les clics de souris ou les entrées au clavier.
- Inconvénients :
  - L'exécution de la requête n'est pas garantie : elle peut échouer si aucun objet ne la traite (ce qui peut être un avantage ou un inconvénient).
  - L'observation des caractéristiques à l'exécution et le débugage peuvent être difficiles..

DP Chain of Responsability : structure

http://www.dofactory.com/Patterns/PatternChain.aspx



DP Chain of Responsability

Étude en autonomie du cours du Site developpez.com sur

le pattern Chain Of Responsability

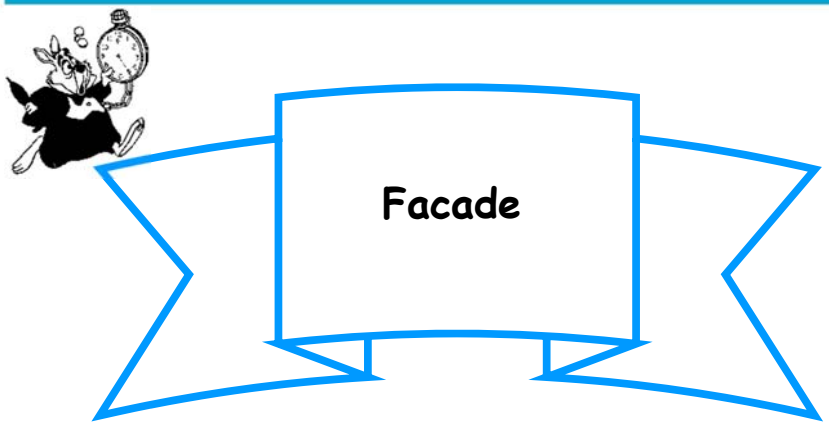
http://rpouiller.developpez.com/tutoriel/java/design-patterns-gang-of-four

Les patterns de structure

4IRC
Version 2011/2012 - Release 2016-2017

F. PERRIN

membre de UNIVERSITÉ DE LYON

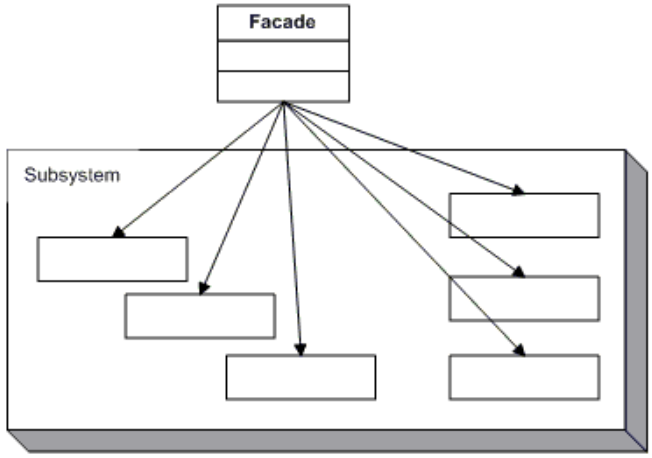


DP Facade enveloppe des objets pour simplifier leur interface

- Quand vous devez simplifier et unifier une grosse interface ou un ensemble d'interfaces complexe, employez une façade.
- Une façade découple un client d'un sous-système complexe.
- L'implémentation d'une façade nécessite de composer la façade avec son sous-système et d'utiliser la délégation pour effectuer le travail de la façade.
- Vous pouvez implémenter plus d'une façade pour un même sous-système.

DP Facade : structure

http://www.dofactory.com/Patterns/PatternFacade.aspx



« Ne parlez pas aux inconnus – Ne parlez qu'à vos amis immédiats »

- Ce principe nous aide à restreindre les interactions entre objets à quelques « amis » proches.
- Lors de la conception d'un système, il signifie que vous devez être attentif pour chaque objet au nombre de classes avec lesquelles il interagit et à la façon dont il entre en interaction avec elles.
- Ce principe nous empêche de créer des systèmes constitués d'un grand nombre de classes fortement couplées dans lesquels les modifications d'une composante se propagent en cascade aux autres parties. Lorsqu'il existe trop de dépendances entre de nombreuses classes, vous construisez un système fragile dont la maintenance sera coûteuse et que les autres auront du mal à comprendre en raison de sa complexité.

Comment ne PAS se faire des amis et influencer les objets

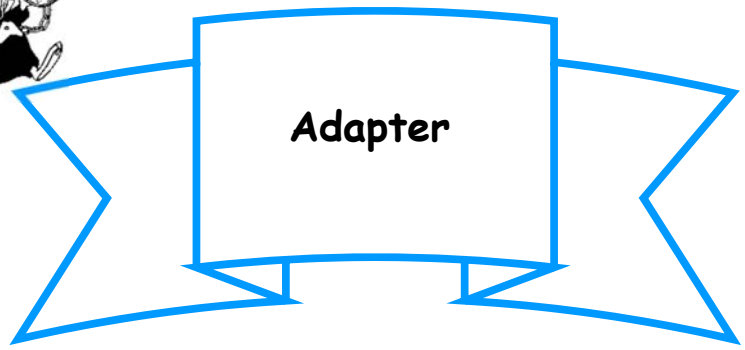
- Comment procéder concrètement ? considérez un objet quelconque ; maintenant, à partir de n'importe quelle méthode de cet objet, le principe nous enjoint de n'appeler que des méthodes qui appartiennent :
  - À l'objet lui-même
  - Aux objets transmis en arguments à la méthode
  - Aux objets que la méthode crée ou instancie
  - Aux composants de l'objet (composant = un objet référencé par une variable d'instance : relation A-UN).
- Ces lignes directrices nous disent de ne pas appeler de méthodes appartenant à des objets retournés par d'autres appels de méthodes !
  - Ex : return station.getThermometre().getTemperature(); Ce code est couplé à 3 classes …

DP Facade

Étude en autonomie du cours du Site developpez.com sur

le pattern Facade

http://rpouiller.developpez.com/tutoriel/java/design-patterns-gang-of-four



DP Adapter permet de faire entrer une cheville ronde dans un trou carré

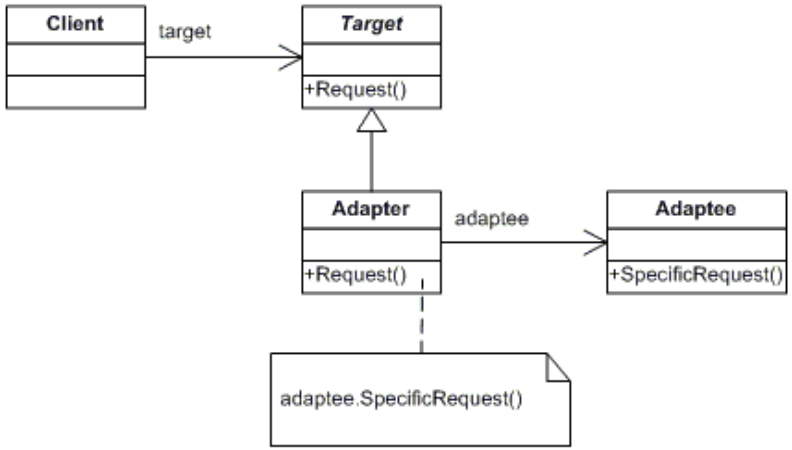
- Quand vous devez utiliser une classe existante et que son interface n'est pas celle dont vous avez besoin, employez un adaptateur.
- Un adaptateur transforme une interface en celle que le client attend.
- L'implémentation d'un adaptateur peut demander plus ou moins de travail selon la taille et la complexité de l'interface cible.

DP Adapter illustre principe de ségrégation des interfaces (ISP)

- « Les clients ne doivent pas être forcés de dépendre d'interfaces qu'ils n'utilisent pas. »
- Lorsqu'une classe est trop riche, tous les clients ont une visibilité sur tous les services rendus par la classe :
  - Chaque client voit une interface trop riche dont une partie ne l'intéresse pas,
  - Chaque client peut subir l'impact des changements d'une interface qu'il n'utilise pas.
- Le pattern adapter est alors une solution.

DP Adapter : structure

http://www.dofactory.com/Patterns/PatternAdapter.aspx



DP Adapter : utilisations remarquables, patterns connexes

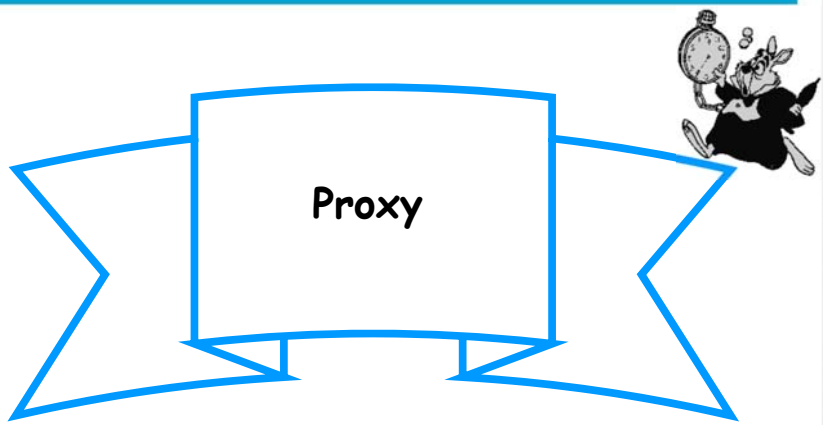
- Utilisations remarquables :
  - Java.util.Arrays#asList().
  - Java.io.InputStreamReader(InputStream) -> retourne un Java.io.Reader.
- Patterns connexes :
  - Toutes les Wrappers qui enveloppent un objet.
  - Cependant, un adaptateur enveloppe un objet pour modifier son interface alors qu'un décorateur enveloppe un objet pour ajouter de nouveaux comportements et de nouvelles responsabilités.

DP Adapter

Étude en autonomie du cours du Site developpez.com sur

le pattern Adapter

http://rpouiller.developpez.com/tutoriel/java/design-patterns-gang-of-four

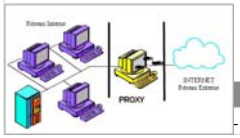


DP Proxy sert d'intermédiaire (1)

- Le pattern Proxy fournit un objet qui se substitut à un autre objet (le sujet) pour contrôler les accès du client.
- Il possède la même interface que le sujet ce qui rend la substitution transparente vis-à-vis des clients.
- Il existe de nombreuses façons de gérer ces accès :
  - Un Proxy Distant gère les interactions entre un client et un objet distant.
  - Un Proxy Virtuel contrôle l'accès à un objet qui est coûteux à instancier.
  - Un Proxy de protection contrôle l'accès aux méthodes d'un objet en fonction de l'appelant.
  - Des proxies de mise en cache, des proxies de synchronisation, des proxies pare-feu, des proxies « copyon-write », etc.

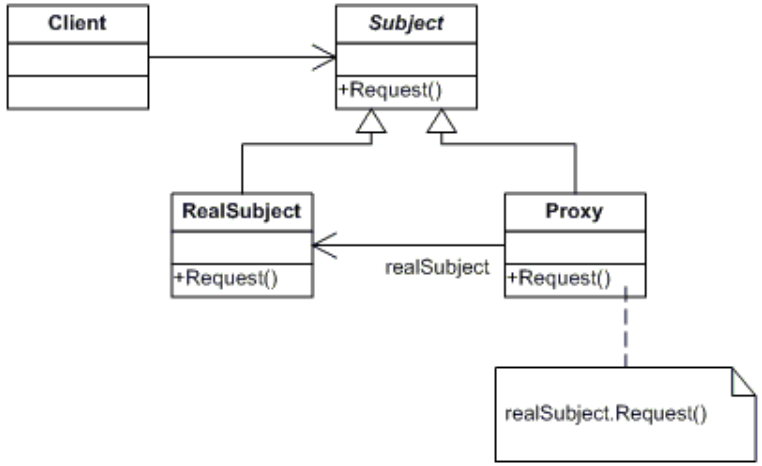
DP Proxy sert d'intermédiaire (2)

- Du point de vue structurel, Proxy est similaire à Décorateur, mais leurs objectifs diffèrent : le pattern Décorateur ajoute un comportement à un objet, alors qu'un Proxy contrôle l'accès.
- Java dispose d'un support intégré de Proxy. Il permet de créer une classe proxy dynamique à la demande et de transmettre tous les appels qui lui sont envoyés à un gestionnaire de votre choix.
- Comme tous les wrappers, les proxies augmentent le nombre de classes et d'objets dans vos conceptions.



DP Proxy : structure

http://www.dofactory.com/Patterns/PatternProxy.aspx

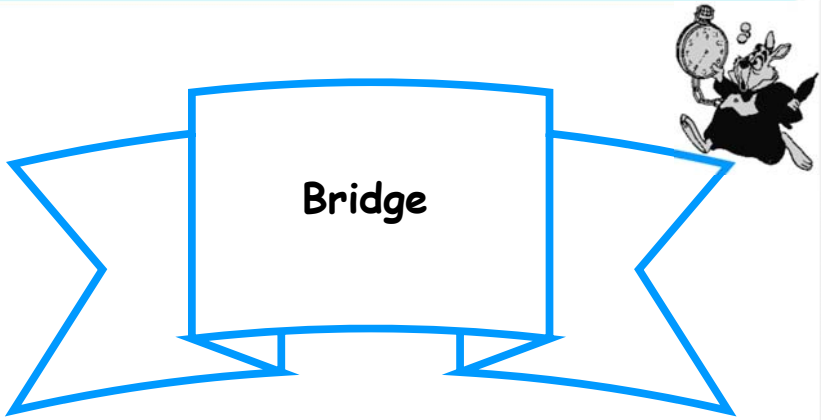


DP Proxy

Étude en autonomie du cours du Site developpez.com sur

le pattern Proxy

http://rpouiller.developpez.com/tutoriel/java/design-patterns-gang-of-four

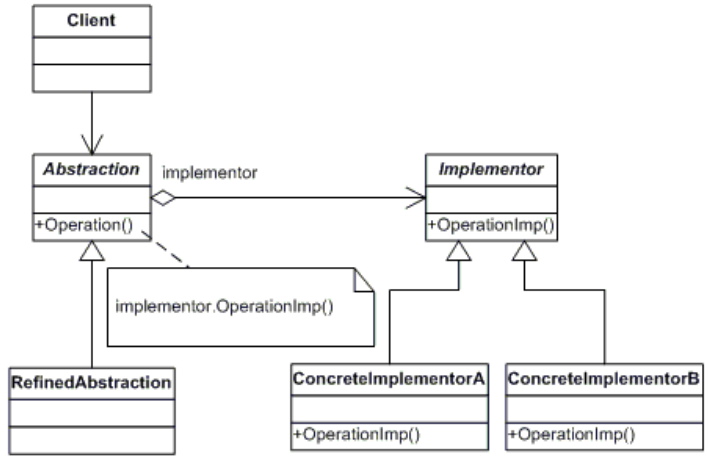


DP Bridge sert à découpler abstraction et implémentation (1)

- Découple l'aspect d'implémentation d'un objet de son aspect de représentation et d'interface.
- L'implémentation est complètement encapsulée ce qui permet que l'implémentation et sa représentation puisse évoluer indépendamment sans que l'une n'exerce une contrainte sur l'autre.
- Ni les modifications des classes concrètes de l'abstraction, ni bien sûr de l'implémentation, n'affectent le client.

DP Bridge : structure

http://www.dofactory.com/net/bridge-design-pattern



DP Bridge

Étude en autonomie du cours du Site developpez.com sur le pattern Bridge

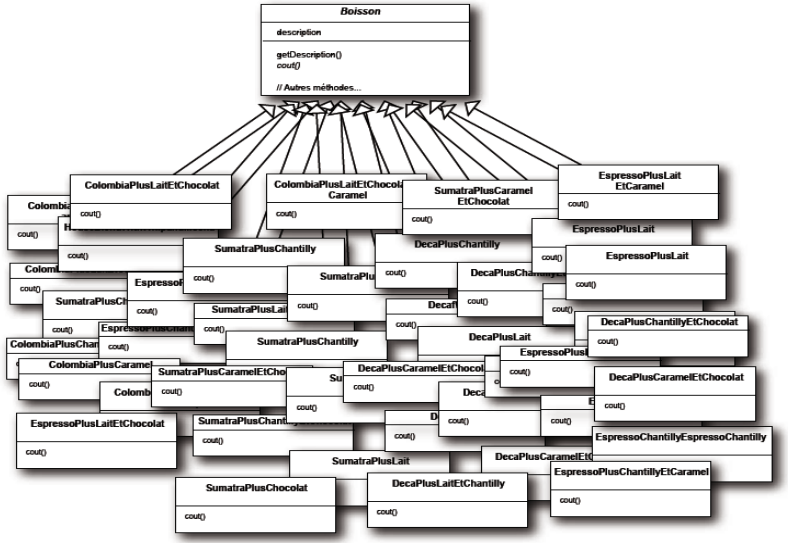
http://rpouiller.developpez.com/tutoriel/java/design-patterns-gang-of-four

Decorator

Métaphore : calcul du prix du café

- Énoncé :
  - Un café propose diverses variétés de café : colombia, déca, expresso, etc. Pour satisfaire leurs clients il leur propose des suppléments chocolats, sirop de caramel, le tout couronné de chantilly, etc.
  - Chaque café et chaque ingrédient supplémentaire possède un prix fixe. Le coût du café final sera la somme des prix de ses composants.

1ère approche : 1 classe pour chaque type de combinaison



Que faire des éléments qui varient ? (1)

- Nous constatons que :
  - Le nombre de classe est exponentiel. Que se passe-t-il si on propose un nouvel ingrédient (vanille) ?
  - Chaque fois qu'il y a un changement (prix du lait) il faut modifier de nombreuses classes.

➡ L'héritage n'est sans doute pas la bonne solution.

Que faire des éléments qui varient ? (3)

Problème :

Comment étendre facilement les classes pour incorporer de nouveaux comportements sans modifier le code existant.

Solution :

Rappelez vous le pouvoir de la composition !

Comment le DP Decorator résout-il le Pb ? (1)

- Le DP Decorator fournit une solution de rechange à l'héritage et emploie la composition et la délégation pour ajouter des comportements au moment de l'exécution.
- Il fait appel à un ensemble de classes « décorateurs » que l'on utilise pour envelopper des composants concrets. Un composant peut être enveloppé dans un nombre quelconque de décorateurs.
- L'utilisation de décorateurs est transparente vis-à-vis des clients.

Comment le DP Decorator résout-il le Pb ? (2)

- Les classes « décorateurs » reflètent le type des composants qu'elles décorent : les décorateurs et les objets destinés à être décorés doivent dériver d'une même super-classe…
- Les objets décorateurs ont une instance du type de leur super-classe en leur sein. Grâce à cette instance, ils peuvent invoquer la méthode commune et ainsi rajouter des traitements à celle-ci.

Étudions la solution (1)

- Objectif des décorateurs : ajouter du comportement aux Jcomponent du package javax.swing.

```
class Decorator extends JComponent {
    public Decorator(JComponent c) {
        super();
        setLayout(new BorderLayout());
        add("Center", c);
    }
}
```


Étudions la solution (2)

```
class CoolDecorator extends Decorator {
    // …

    public CoolDecorator(JComponent c) {
        super(c);
        this.repaint();
        // …
    }

    public void paint(Graphics g) {
        super.paint(g); //first draw the parent Jcomponent
        // Nouvelles fonctionalités du paint()
    }
}
```


Étudions la solution (3)

```
public class DecoWindow extends JFrame  {

    public DecoWindow() {
        super("Deco Button");
        JPanel jp = new JPanel();
        jp.add(new CoolDecorator(new JButton("Cool")));
        jp.add(new SlashDecorator(new CoolDecorator
                                  (new JButton("CoolSlash"))));
        // …
    }
    static public void main(String argv[]) {
        new DecoWindow();
    }
}
```


Étudions la solution (4)

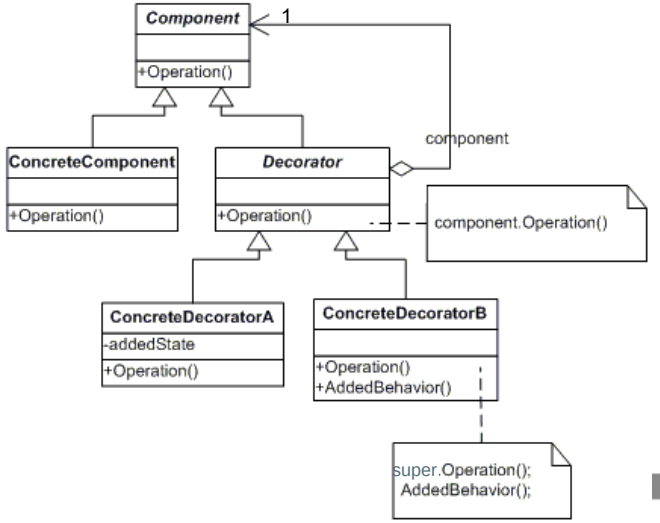
- Les décorateurs ont le même supertype que les objets qu'ils décorent (`JComponent`).
- Vous pouvez utiliser un ou plusieurs décorateurs pour envelopper un objet.
- Comme le décorateur a le même supertype que l'objet qu'il décore, nous pouvons transmettre un objet décoré à la place de l'objet original (enveloppé).
- Le décorateur ajoute son propre comportement soit avant soit après avoir délégué le reste du travail à l'objet qu'il décore.
- Les objets pouvant être décorés à tout moment, nous pouvons les décorer dynamiquement au moment de l'exécution avec autant de décorateurs que nous en avons envie.

DP Decorator : classification, intention, utilisation

- Classification : pattern structurel.
- Intention :
  - Le DP Decorator attache des responsabilités supplémentaires à un objet de façon dynamique.
- Indication d'utilisation lorsque :
  - Un système ajoute/retire dynamiquement des fonctionnalités à un objet, sans que les clients de l'objet soit modifié.
  - L'utilisation de sous-classe pour enrichir un objet génère une hiérarchie de classes très complexe ou est impossible (classe finale).

DP Decorator : structure

http://www.dofactory.com/Patterns/PatternDecorator.aspx



DP Decorator : participants

- Participants :
  - Composant (JComponent) est l'interface commune aux composants et aux décorateurs.
  - ComposantConcret (JButton) est l'objet initial auquel les nouvelles fonctionnalités vont être ajoutées.
  - Decorateur (Decorator) est une classe abstraite qui détient une référence vers un Composant.
  - DecorateurConcrets (SlashDecorator) implémentent les fonctionnalités ajoutées au composant.

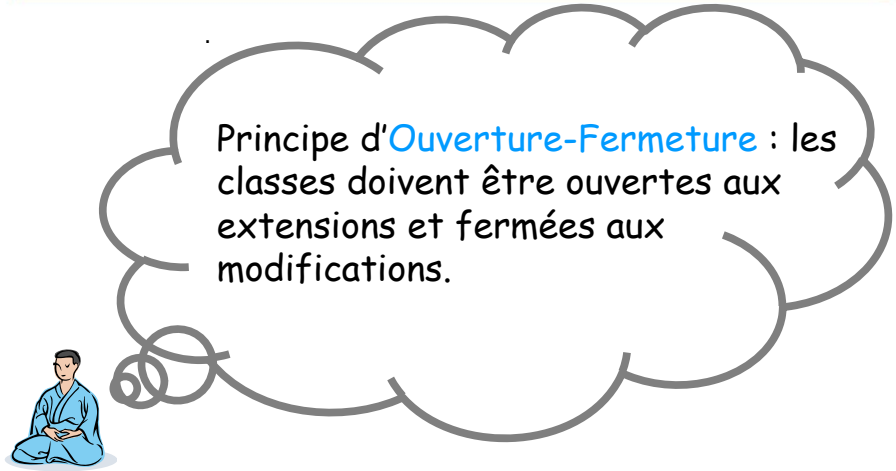
DP Decorator : collaborations, conséquences

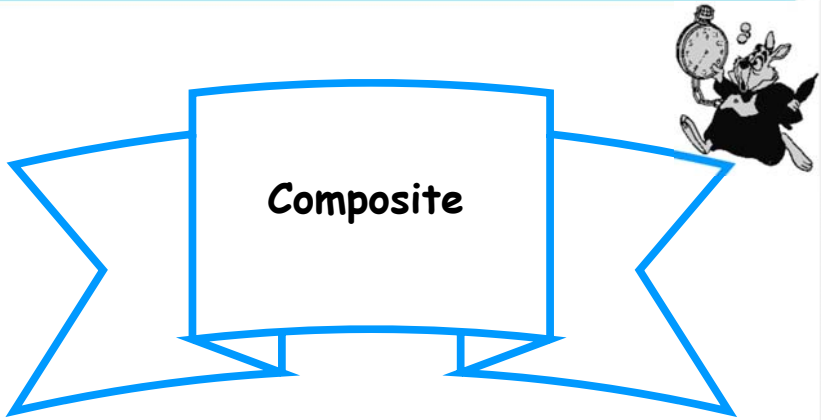
- Collaborations :
  - Le décorateur se substitue au composant.
  - Lorsqu'il reçoit un message destiné à ce dernier, il le redirige au composant en effectuant des opérations préalables ou postérieures à cette redirection.
- Conséquences :
  - Les décorateurs sont transparents pour les clients du composant sauf si le client dépend du type concret du composant.
  - Les décorateurs peuvent entrainer la création de nombreux petits objets.

DP Decorator : utilisations remarquables, patterns connexes

- Utilisations remarquables :
  - La gestion des Entrées/Sorties dans java.io.
  - Java.util.Collections#checkedXXX()
- Patterns connexes :
  - Composite : les Decorator sont conçu comme des Composites ne contenant qu'1 seul élément. Cependant, un Composite ne met pas en valeur l'ajout dynamique comme le fait le Decorator.
  - Proxy : l'emballage et l'objet emballé partagent la même interface de manière à ce que le Proxy puisse se substituer au sujet réel. Les Proxy contrôlent l'accès au sujet réel.

Principes objets illustrés par ce patterns



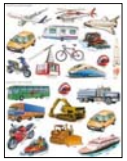


Métaphore : recette de cuisine

- Une recette se compose d'une liste d'ingrédients.
- Certains sont des éléments atomiques (lait, farine, etc.).
- D'autres sont composites (béchamel, etc.).
- Et pourtant tous rentrent dans la composition de la recette de la même manière (ajouter, mélanger, cuire, etc.).

Pb : offre de maintenance d'un parc de véhicules

- Dans un systèmes de vente de véhicule, nous voulons connaitre le nb de véhicule dont disposent les sociétés clientes et leur proposer des offres de maintenance de leur parc.
- Les sociétés qui possèdent des filiales demandent des offres de maintenance qui prennent en compte le parc de leurs filiales.
- Une solution immédiate consiste à traiter différemment les sociétés sans et avec filiales.



Que faire des éléments qui varient ?

- Nous constatons que :
  - Notre client devrait utiliser l'opérateur instanceof pour tester le type de la société, utilisation contre laquelle nous luttons farouchement !
  - On ne respecterait plus le principe « programmez des interfaces et non des implémentations » ce qui rendrait l'application plus complexe et fermée à l'extension.

Que faudrait-il faire pour résoudre ce Pb ?

Problème :
Comment traiter de la même façon des objets atomiques (feuilles) et des objets composites contenant d'autres objets ?

Solution :
Faire implémenter la même interface aux objets feuilles et aux objets composites.

Comment le DP Composite résout-il le Pb ?

- Le pattern Composite résout ce Pb en unifiant l'interface des 2 types de société et en utilisant la composition récursive.
- Cette composition récursive est nécessaire car une société peut posséder une filiale qui elle-même peut posséder des filiales.
- Le DP Composite introduit donc la notion de composant : un composant est soit un composite (composé de composants), soit une feuille.
- Les clients pourront ainsi traiter tous les composants de la même manière.

Étudions la solution (1)

Répertoire « Composite » [2]

// Composant

```
public abstract class Societe  {
        protected static double coutUnitVehicule = 5.0;
        protected int nbrVehicules;
        public void ajouteVehicule()   {
            nbrVehicules = nbrVehicules + 1;
        }
        public abstract double calculeCoutEntretien();
        public abstract boolean ajouteFiliale(Societe filiale);
}
```


Étudions la solution (2)

// Composite ou Composé

```
public class SocieteMere extends Societe {
  protected List<Societe> filiales = new ArrayList<Societe>();
  public boolean ajouteFiliale(Societe filiale)   {
        return filiales.add(filiale);
  }
  public double calculeCoutEntretien()   {
        double cout = 0.0;
        for (Societe filiale : filiales)
              cout = cout + filiale.calculeCoutEntretien();
        return cout + nbrVehicules * coutUnitVehicule;
  }
}
```


Étudions la solution (3)

// Feuille

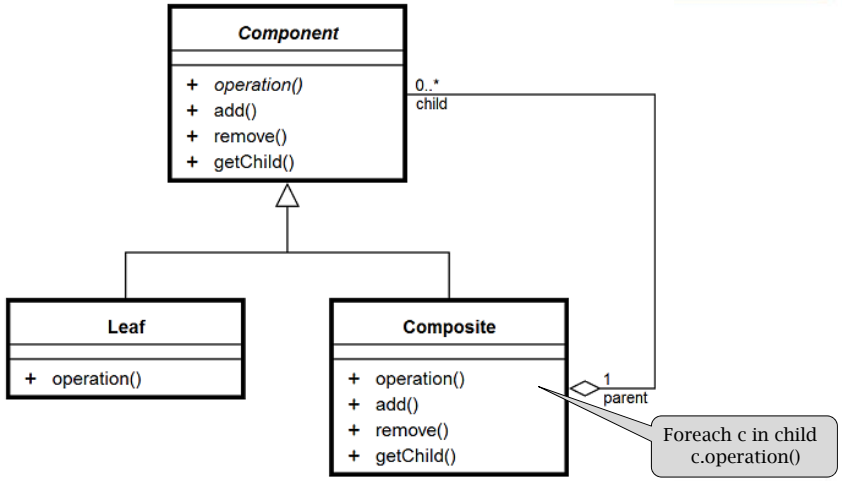
```
public class SocieteSansFiliale extends Societe {
  public boolean ajouteFiliale(Societe filiale)   {
        return false;
  }
  public double calculeCoutEntretien()  {
        return nbrVehicules * coutUnitVehicule;
  }
}
```


DP Composite : classification, intention, utilisation

- Classification : pattern structurel.
- Intention :
  - DP 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.
- Indication d'utilisation lorsque :
  - Il est nécessaire de représenter au sein d'un système des hiérarchie de composition.
  - Les clients d'une composition doivent ignorer s'ils communiquent avec des objets composés ou non.

DP Composite : structure

http://cyrille.giquello.fr/informatique/design_pattern



DP Composite : participants,

- Participants :
  - Composant (Societe) est la classe abstraite qui introduit l'interface des objets de la composition, implémente les méthodes communes et introduit la signature des méthodes qui gèrent la composition (ajout, suppression composants).
  - Feuilles (SocieteSansFiliale) est la classe concrète qui décrit les objets non composés (feuilles de l'arbre).
  - Composé (SocieteMere) est la classe concrète qui décrit les objets composés de la hiérarchie. Elle possède une relation d'agrégation avec la classe Composant.

DP Composite : collaborations

- Collaborations :
  - Les clients envoient leurs requêtes aux composants au travers de l'interface de la classe Composant.
  - Lorsqu'un Composant reçoit une requête, il réagit en fonction de sa classe :
    - Si c'est une Feuille, il traite la requête simplement.
    - Si c'est un Composé :
      - Il effectue un traitement préalable.
      - Envoie un message à chacun de ses composants.
      - Effectue un traitement postérieur.

DP Composite : conséquences (1)

- Conséquences :
  - Comme les clients n'ont pas besoin de savoir s'ils ont affaire à un objet composite ou à un objet feuille, ils n'ont pas besoin d'écrire des instructions `if` pour être sûrs d'appeler les bonnes méthodes sur les bons objets.
  - Ils peuvent souvent effectuer un seul appel de méthode et exécuter une opération sur toute une structure.

DP Composite : conséquences (2)

- Cette transparence vis-à-vis des clients a un défaut : on ne respecte pas le principe de responsabilité unique :
  - Nous avons 2 types d'opérations dans un composant : les opérations de gestion des enfants ET les opérations des feuilles.
  - Nous y perdons quelque peu en sécurité parce qu'un client peut essayer d'appliquer une opération inappropriée ou sans objet à un élément.
  - Dans les méthodes non applicables cependant héritées par les feuilles (add, remove…) il suffit alors de lever une exception :

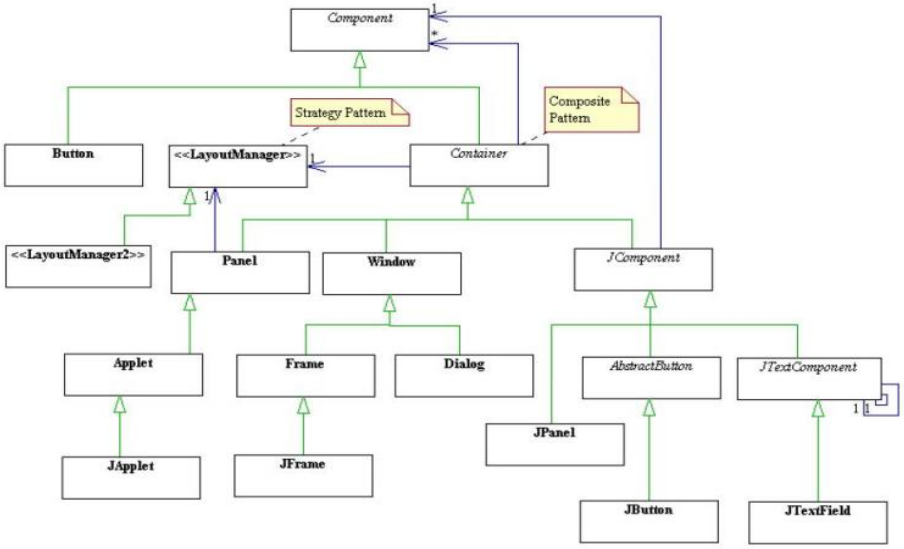
```
throw new UnsupportedOperationException();
```


DP Composite : utilisations remarquables, patterns connexes

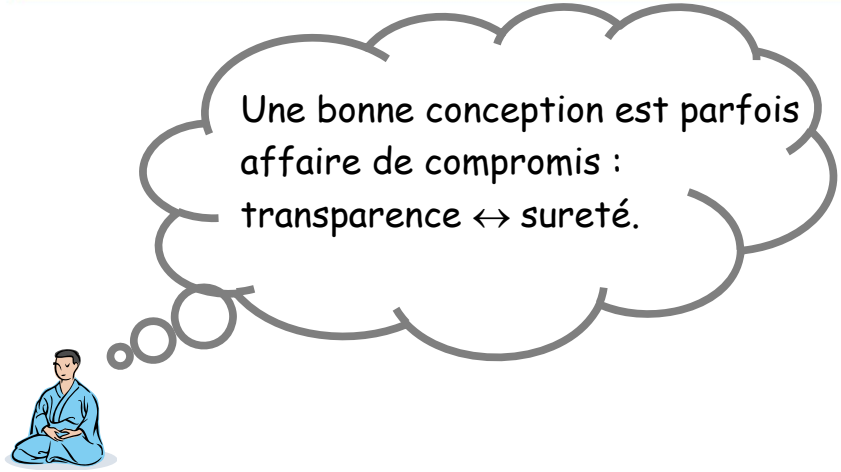
- Utilisations remarquables :
  - API Graphique de Java : javax.swing.Jcomponent#add(Component).
  - Java.util.List#addAll(Collection).
- Patterns connexes :
  - Decorator : comme Composite, ce pattern fournit la même interface pour l'élément qu'il enveloppe.
  - Iterator : utilisé pour parcourir une structure composite.
  - Visitor : utilisé pour parcourir un composite et propager un traitement.

Les DP Composite et Strategy sont utilisés pour gérer les composants graphiques en Java

www.siteduzero.com



Principes objets illustrés par ce patterns



Les patterns de création

4IRC
Version 2011/2012 - Release 2016-2017

F. PERRIN

membre de Université de Lyon

CPE Lyon

Singleton

DP Singleton empêche la multiplication des objets

- Le Pattern garantit qu'il existe au plus une instance d'une classe donnée dans une application. Le Pattern Singleton fournit également un point d'accès global à cette instance.
- Java implémente le Pattern Singleton en utilisant un constructeur privé, une méthode statique combinée à une variable statique.
- Examinez vos contraintes de performances et de ressources et choisissez soigneusement une implémentation de Singleton appropriée pour les applications multithread (et il faut considérer que toutes les applications sont multithread).

DP Singleton : structure

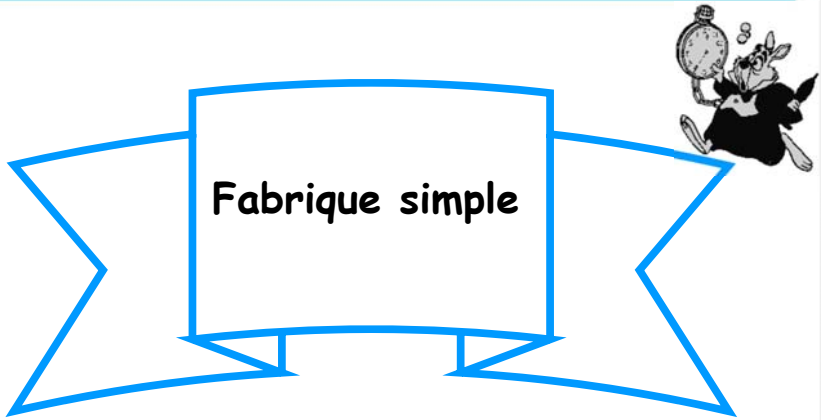
http://www.dofactory.com/Patterns/PatternSingleton.aspx

Etudions la solution – attention risque en cas de multi-thread

```
public final class Singleton {
    private static Singleton instance;

    private Singleton() {          // private empêche new
    }
    public static Singleton getInstance() {
        if (instance == null)
                instance = new Singleton();
        return instance;
    }
    public Object clone() throws CloneNotSupportedException {
        throw new CloneNotSupportedException();
    }
}
```

Etudions la solution threadsafe

```
public final class MonSingleton {

    // initialisation effectuée au chargement de la classe en mémoire
    private static MonSingleton instance =
                    new MonSingleton();

    private MonSingleton() { }

    public static MonSingleton getInstance() {
        return instance;
    }
}
```



Pb : gestion d'un ensemble de pizzerias vendant des pizzas à emporter

- Lorsque vous avez tout un ensemble de classes concrètes apparentées, le code peut ressembler à ça :

```
Pizza commanderPizza(String type) {
    Pizza pizza;
    if (type.equals("fromage")) {
      pizza = new PizzaFromage();
      } else if (type.equals("poivrons") {
            pizza = new PizzaPoivrons();
      } else  // …
    }
    pizza.preparer();
    pizza.cuire();
    pizza.couper();
    pizza.emballer();
    return pizza;
}
```

Que faire des éléments qui varient ? (1)

- Nous constatons que :
  - Nous avons plusieurs instanciations de classes concrètes et la décision de la classe à instancier est prise au moment de l'exécution, en fonction d'un certain nombre de conditions.
  - Lorsqu'il faudra apporter des modifications , des extensions ou des suppressions, vous devrez reprendre ce code.
  - Ce type de code se retrouve souvent dans plusieurs parties de l'application, ce qui rend la maintenance et les mises à jour plus difficiles et plus sujettes à l'erreur.
  - Votre code n'est donc pas « fermé à la modification ».
  - Utiliser new revient à programmer une implémentation non une abstraction.
  - Si votre code s'appuie sur une interface, il fonctionnera avec toute nouvelle classe qui implémente cette interface via le polymorphisme.

Que faire contre les Pb de couplage fort liés à l'instanciation ?

L'instanciation est une activité qui ne devrait pas toujours être publique et qui peut souvent entraîner des problèmes de couplage.
Les patterns fabriques peuvent vous aider à vous libérer de dépendances embarrassantes.

Comment la Fabrique simple résout-elle le Pb ?

- Il suffit d'encapsuler les new dans un autre objet : la fabrique simple.
- Le client (la pizzeria) passe alors par la fabrique pour obtenir un Produit concret (pizzaFromage) qui étend le produit abstrait (Pizza).
- Tout l'intérêt, c'est qu'une fabrique simple peut avoir plusieurs clients :
  - Classe CartePizzas qui utiliserait la fabrique pour accéder à la description et au prix des pizzas.
  - Classe LivraisonADomicile qui gérerait les pizzas différemment de notre classe Pizzeria mais qui serait également un client de la fabrique.
  - etc.

Étudions la solution (1) Répertoire « fabriques/pizzas » [1]

```
public class SimpleFabriqueDePizzas {
    public Pizza creerPizza(String type) {
        Pizza pizza = null;
        if (type.equals("fromage")) {
            pizza = new PizzaFromage();
        }
        else if (type.equals("poivrons")) {
            pizza = new PizzaPoivrons();
        }
        else // …
        return pizza;
    }
}
```

Étudions la solution (2)

```
public class Pizzeria {
    SimpleFabriqueDePizzas fabrique;
    public Pizzeria(SimpleFabriqueDePizzas fabrique) {
        this.fabrique = fabrique; }
    public Pizza commanderPizza(String type) {
        Pizza pizza;
        pizza = fabrique.creerPizza(type);
        pizza.preparer();
        pizza.cuire();
        pizza.couper();
        pizza.emballer();
        return pizza;
    }  //  + autres méthodes
}
```


Étudions la solution (3)

```
public class Application{
    // ...
    SimpleFabriqueDePizzas fabriquePizza = new
        SimpleFabriqueDePizzas ();
    Pizzeria maBoutique = new Pizzeria(fabriquePizza);
    maBoutique.commanderPizza("fromage");
    // …
}
```


Factory Method

Pb : gérer des fabriques dans des pizzerias différentes

- Il peut exister des pizzeria dans différentes régions qui souhaitent des déclinaisons différentes de la pate (fine ou soufflée), du type de sauce (légère ou relevée), de la quantité de fromage, etc.

  ⇒ Pas de PB, on va créer autant de fabriques simples que de pizzéria.

- Cependant, on ne souhaite pas que les pizzérias soient créatives sur le process : temps de cuisson variable, non découpage de la pizza avant emballage, etc.

  Une seule solution (rappelez-vous le DP Template Method) :

  ⇒
  - L'algorithme du process doit être imposé par la classe Pizzeria,
  - la méthode de création sera abstraite et
  - sera implémentée par les classes des pizzerias régionales qui appelleront la bonne fabrique.

Comment le DP Factory Method résout-il le Pb ? (1)

- Une méthode de fabrique gère la création des objets et l'encapsule dans une sous-classe. Cette technique découple le code client de la superclasse (Pizzeria) du code de création des objets implémenté dans la sous-classe (PizzeriaBrest).
- La superclasse Pizzeria définit une méthode de fabrique abstraite creerPizza() que les sous-classes (PizzeriaBrest) implémentent pour produire des produits.

- Chaque boutique est libre de créer son propre style de pizza en implémentant creerPizza().
- creerPizza() permet alors d'instancier des produits concrets (PizzaPoivronsStyleBrest).

Étudions la solution (1) Répertoire « fabriques/pizzafm » [1]

```
public abstract class Pizzeria {    // Créateur abstrait
    public final Pizza commanderPizza(String type) {
        Pizza pizza;
        pizza = creerPizza(type);
        pizza.preparer();
        pizza.cuire();
        pizza.couper();
        pizza.emballer();
        return pizza;
    }
    protected abstract Pizza creerPizza(String type);
    // autres méthodes
}
```


Étudions la solution (2)

```
public class PizzeriaBrest extends Pizzeria {        // Créateur concret
    Pizza creerPizza(String item) {
        if (choix.equals("fromage")) {
            return new PizzaFromageStyleBrest();
        } else if (choix.equals("vegetarienne")) { // … }
    }
}
public class PizzaTestDrive {
    public static void main(String[] args) {
        Pizzeria boutiqueBrest = new PizzeriaBrest();
        Pizza pizza = boutiqueBrest.commanderPizza("fromage");
        // …

}
```


Principe d'Inversion des Dépendances (1)

- Le principal problème de la Pizzeria hyper dépendante (début du chapitre) est qu'elle dépend de chaque type de pizza (PizzaFromage, PizzaPoivron…) parce qu'elle instancie les types concrets dans sa méthode commanderPizza().
- Si nous avions bien créé une abstraction, Pizza, nous créions néanmoins les pizzas concrètes dans ce code, ce qui nous empêchait d'exploiter réellement les avantages de cette abstraction.
- Le pattern Factory Method nous permet d'extraire ces instanciations de la méthode commanderPizza()et de le reporter dans les créateurs concrets (classes dérivées).

Principe d'Inversion des Dépendances (2)

- Le Principe d'Inversion des Dépendances nous suggère que nos composants de haut niveau ne doivent pas dépendre des composants de bas niveau, mais que les deux doivent dépendre d'abstractions.
- Après avoir appliqué Factory Method , nous pouvons remarquer que notre composant de haut niveau, Pizzeria, et nos composants de bas niveau, les pizzas, dépendent tous de Pizza, l'abstraction.
- Factory Method n'est pas la seule technique qui permet de respecter le Principe d'Inversion des Dépendances, mais c'est l'une des plus puissantes.

Principe d'Inversion des Dépendances (3)

- Les lignes directrices suivantes peuvent vous aider à éviter les conceptions OO qui enfreignent le Principe d'inversion des dépendances :
  - Aucune variable ne doit contenir une référence à une classe concrète : utilisez une fabrique.
  - Aucune classe ne doit dériver d'une classe concrète : sous-classez une abstraction.
  - Aucune classe ne doit redéfinir une méthode implémentée dans une classe de base, (si ce n'est pour l'enrichir), sinon :
    - La classe de base n'était pas une abstraction.
    - Les méthodes déjà implémentée dans une classe de base sont connues pour être partagées par toutes les sous-classes.

Principe d'Inversion des Dépendances (4)

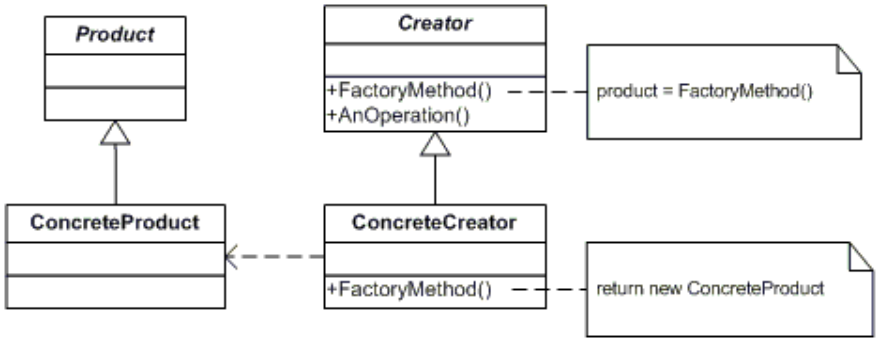
- Développeur sceptique : « Mais, attendez… Ces conseils sont impossibles à suivre. Si je les applique tous, je n'écrirai jamais un seul programme ! »
- Gourou bienveillant : « Comme pour beaucoup de nos principes, il s'agit de lignes directrices vers lesquelles vous devez tendre et non de règles que vous devez appliquer tout le temps ».

DP Factory Method : classification, intention, utilisation

- Classification : pattern créateur.
- Intention :
  - Le pattern Factory Method introduit une méthode abstraite de création d'un objet en déléguant aux sous-classes concrètes la création effective.
- Indication d'utilisation lorsque :
  - Une classe ne connait que les classes abstraites des objets avec lesquelles elle possède des relations.
  - Une classe veut transmettre à ses sous-classes les choix d'instantiation en profitant du mécanisme du polymorphisme.

DP Factory Method : structure

http://www.dofactory.com/Patterns/PatternFactory.aspx



DP Factory Method : participants

- Participants :
  - Créateur (Pizzeria) : classe abstraite qui contient les implémentations de toutes les méthodes destinées à manipuler les produits, et qui invoquent la méthode de fabrication.
  - CréateurConcret (PizzeriaBrest) : classe concrète dont le seul but est d'implémenter la méthode de fabrique. Il peut exister plusieurs créateurs concrets.
  - Produit (Pizza) : classe abstraite décrivant les propriétés communes des produits.
  - ProduitConcret (PizzaFromageStyleBrest) : classe concrète décrivant complètement un produit.

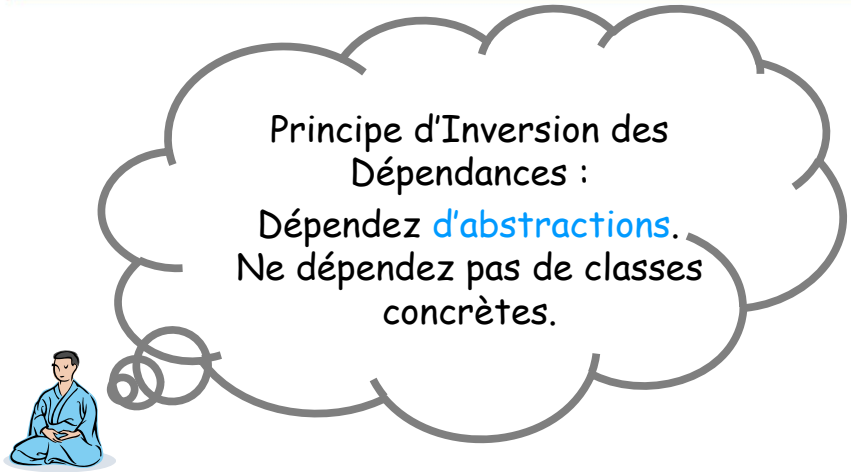
DP Factory Method : collaborations, conséquences

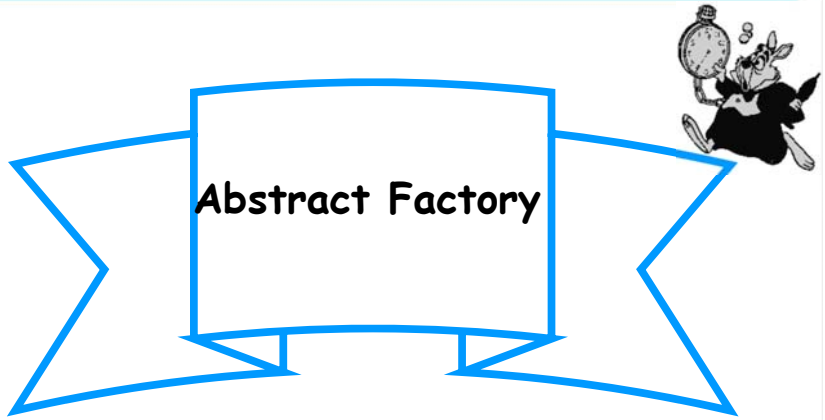
- Collaborations :
  - Les méthodes concrètes de la classe CréateurAbstrait se basent sur l'implémentation de la méthode fabrique dans les sous-classes.
  - Cette implémentation crée une instance de la sous-classe adéquate de Produit.
- Conséquences :
  - Cette technique découple le code client de CréateurAbstrait du code de création des objets de CréateurConcret : les méthodes de la classe CréateurAbstrait savent qu'elle savent manipuler des Produits, sans aucun indice sur le produit réellement créé.

DP Factory Method : utilisations remarquables, patterns connexes

- Utilisations remarquables :
  - java.lang.Object#toString().
  - Java.lang.Class#newInstance().
- Patterns connexes :
  - Template Method : même Principe d'Inversion des Dépendances.
  - Abstract Factory : Dans une fabrique abstraite, les méthodes pour créer les produits sont souvent implémentés à l'aide d'une fabrication (Factory Method).

Principes objets illustrés par ce patterns





Pb : gérer les différents ingrédients sur les pizzas de chaque pizzeria

- Les pizzerias régionales respectent bien la procédure de commande, mais qu'en est-il de la qualité et du nombre des ingrédients lors de la préparation des pizzas, celle-ci étant sous la responsabilité de chaque pizza concrète d'une pizzeria ?

  Rq : par ailleurs, le nb de pizza concrètes est assez impressionnant…

  ➡ La solution est de leur imposer un cadre pour la préparation des pizzas et la gestion des ingrédients.

- Toutes les pizzas sont fabriquées à partir des mêmes composants, mais chaque région a une implémentation différente de ces composants : pate (fine ou soufflée), type de sauce (légère ou relevée), type et quantité de fromage, etc.

  ➡ Comment gérer les familles de composants ? En construisant une fabrication responsable de la création de chaque ingrédient de la famille (pate, sauce, fromage, etc.).

Comment le DP Abstract Factory résout-il le Pb ? (1)

- Nous créons d'abord une interface pour la fabrication qui va créer toutes nos garnitures (FabriqueIngredientsPizza).
- Puis nous construisons une fabrique pour chaque région : classes qui implémentent chaque méthodes de création de cette interface (fabriqueIngredientspizzaBrest).
- Nous implémentons ensuite un ensemble de classes ingrédients qui seront utilisées avec la fabrique, comme Reggiano, PoivronsRouges et PateSoufflee. Ces classes peuvent être partagées entre les régions en fonction des besoins.
- La méthode preparer() de la classe Pizza est revisitée.
- Puis nous assemblons le tout en incorporant nos fabriques dans l'ancien code des pizzerias concrètes.

Étudions la solution (1)

```
public interface FabriqueIngredientsPizza {
    public Pate creerPate();
    public Sauce creerSauce();
    public Fromage creerFromage();
    public Legume[] creerLegumes();
    public Poivrons creerPoivrons();
    public Moules creerMoules();
}
```

Étudions la solution (2)

```
public class FabriqueIngredientsPizzaBrest implements
                                        FabriqueIngredientsPizza {
    public Pate creerPate() { return new PateFine(); }
    public Sauce creerSauce() { return new SauceMarinara(); }
    public Fromage creerFromage() { return new Reggiano(); }
    public Legume[] creerLegumes() {
        Legume legumes[] = { new Ail(), new Oignon(), new
        Champignon(), new PoivronRouge() };
        return legumes;
    }
    public Poivrons creerPoivrons() { return new PoivronsEnRondelles(); }
    public Moules creerMoules() { return new MoulesFraiches(); }
}
```


Étudions la solution (3)

```
public interface Pate {
    public String toString();
}

public class PateFine implements Pate {
    public String toString() {
        return "Pâte fine";
    }
}
```

Et etc. pour chaque ingrédient.

Étudions la solution (4)

```
public abstract class Pizza {

    String nom;   Pate pate;        Sauce sauce;    Legume legumes[];
    Fromage fromage;    Poivrons poivrons;        Moules moules;

    abstract void preparer();
    void cuire() {  // … }
    void couper() { // … }
    void emballer() { // … }
    void setNom(String nom) { this.nom = nom; }
    String getNom() { return nom;           }
    public String toString() { // …  }
}
```

Étudions la solution (5)

```
public class PizzaFromage extends Pizza {
    FabriqueIngredientsPizza  fabriqueIngredients;

    public PizzaFromage(FabriqueIngredientsPizza fabriqueIngredients) {
        this.fabriqueIngredients = fabriqueIngredients;
    }
    void preparer() {
        System.out.println("Préparation de " + nom);
        pate = fabriqueIngredients.creerPate();
        sauce = fabriqueIngredients.creerSauce();
        fromage = fabriqueIngredients.creerFromage();
    }
}
```


Étudions la solution (6)

```
public abstract class Pizzeria {    // ne change pas p/r exemple précédent

    protected abstract Pizza creerPizza(String item);

    public Pizza commanderPizza(String type) {
        Pizza pizza = creerPizza(type);
        pizza.preparer();
        pizza.cuire();
        pizza.couper();
        pizza.emballer();
        return pizza;
    }
}
```


Étudions la solution (7)

```
public class PizzeriaBrest extends Pizzeria {
    protected Pizza creerPizza(String item) {
        Pizza pizza = null;
        FabriqueIngredientsPizza  fabriqueIngredients =
                new FabriqueIngredientsPizzaBrest();
        if (item.equals("fromage")) {
                pizza = new PizzaFromage(fabriqueIngredients);
                pizza.setNom("Pizza style Brest et fromage");
        } else if (item.equals("vegetarienne")) {
                pizza = new PizzaVegetarienne(fabriqueIngredients);
                pizza.setNom("Pizza végétarienne style Brest");
        } else // …
        return pizza;
    }
}
```


Étudions la solution (8)

```
public static void main(String[] args) {
    Pizzeria pizzeriaBrest = new PizzeriaBrest();
    Pizza pizza = pizzeriaBrest.commanderPizza("fromage");
    System.out.println("Luc a commandé une " + pizza + "\n");
}
```

Étudions la solution (8)

- Récapitulons le déroulement de l'algorithme lors d'une commande :
  - Tout d'abord, il nous faut une PizzeriaBrest :
    - Pizzeria pizzeriaBrest = new PizzeriaBrest();
  - Nous pouvons prendre une commande :
    - pizzeriaBrest.commanderPizza("fromage");
  - La méthode commanderPizza() commence par appeler creerPizza() :
    - Pizza pizza = creerPizza("fromage");
  - Quand la méthode creerPizza() est appelée, c'est là que notre fabrique d'ingrédients (celle de Brest) entre en scène :
    - Pizza pizza = new PizzaFromage(fabriqueIngredients);

Étudions la solution (9)

- Quand la méthode preparer() est invoquée, elle demande à la fabrique de préparer les ingrédients :
  - void preparer() {
    pate = fabriqueIngredients.creerPate();
    sauce = fabriqueIngredients.creerSauce();
    fromage = fabriqueIngredients.creerFromage();
    }
- Enfin, nous avons la pizza préparée en main et la méthode commanderPizza() la fait cuire, la découpe et l'emballe.

DP Abstract Factory : classification, intention

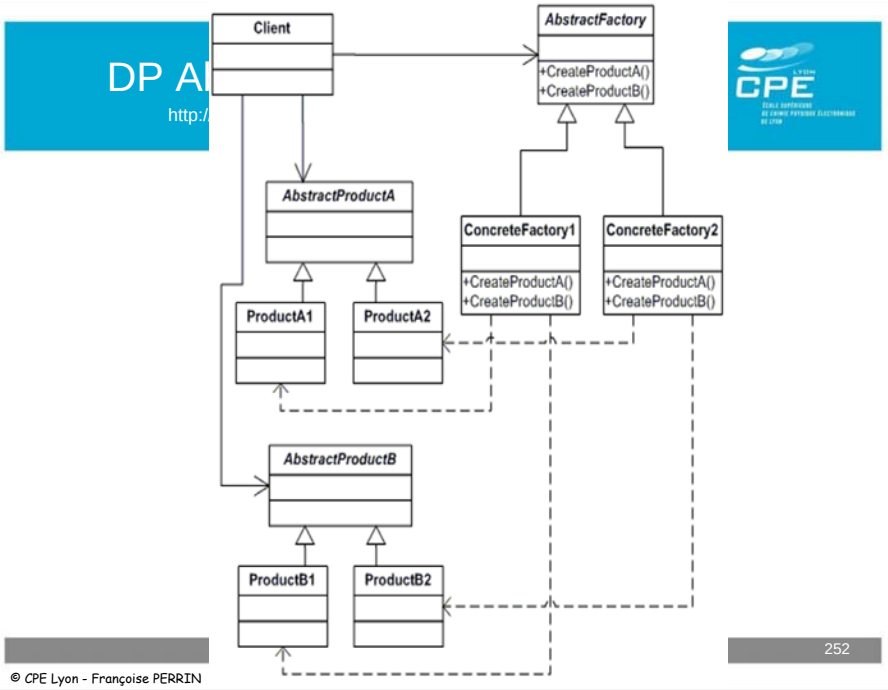
- Classification : pattern créateur.
- Intention :
  - Abstract Factory fournit une interface pour créer des familles d'objets apparentés sans avoir à dépendre de leur classe concrète.
  - Abstract Factory s'appuie sur la composition : la création des objets est implémentée dans les méthodes exposées dans l'interface FabriqueAbstraite.

DP Abstract Factory : utilisation

- Indication d'utilisation lorsque :
  - Un système doit être indépendant de la façon dont ces produits sont crées et composés.
  - Un système est paramétré avec un produit issu d'une famille de produits (qui peuvent évoluer).

DP Al

http://



DP Abstract Factory : participants

- Participants :
  - FabriqueAbstraite (FabriqueIngredientsPizza) est une interface spécifiant les signatures des méthodes créant les différents produits (Pate, Sauce, etc.).
  - FabriqueConcrete 1&2 (FabriqueIngredientsPizzaBrest) sont les classes qui implémentent les méthodes créant les ProduitsConcrets (PateFine, SauceMarinara, etc.) pour chaque famille de produits (les ingrédients).
  - ProduitConcret 1&2 sont les classes des produits créés. Il existent cependant indépendamment des fabriques et peuvent être utilisés par plusieurs fabriques.
  - Client (Pizza) est la classe qui utilise l'interface de FabriqueAbstraite. La Pizza ainsi créée délègue à la fabrique la création des ingrédients

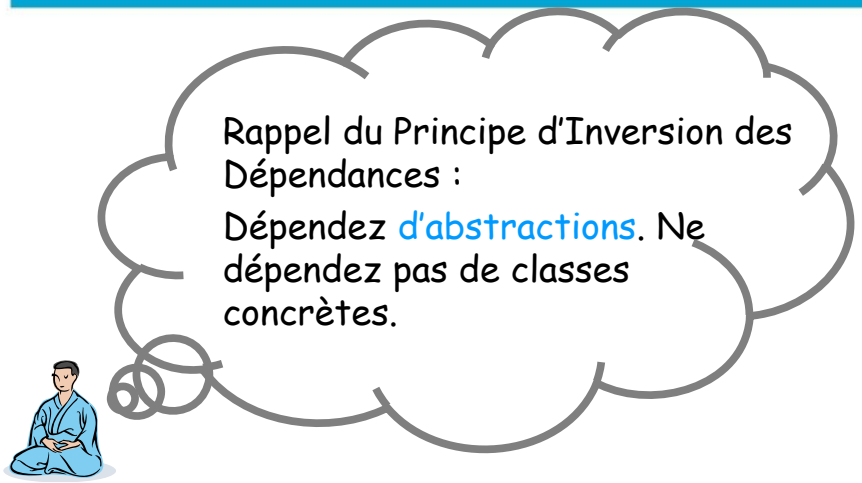
DP Abstract Factory : collaborations, conséquences

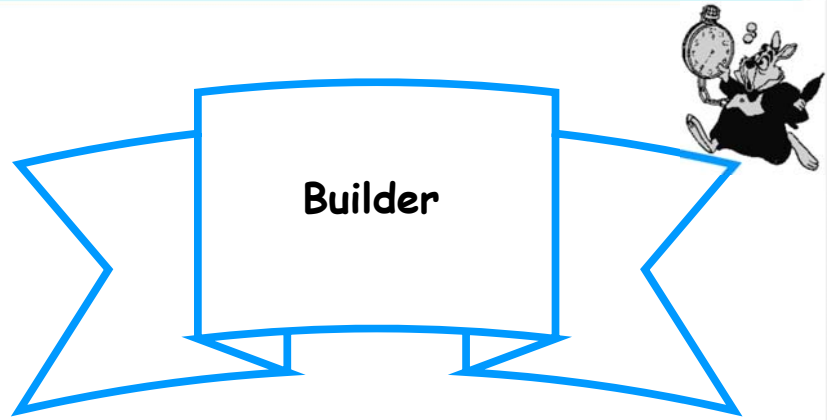
- Collaborations :
  - La classe Client est écrite pour utiliser la FabriqueAbstraite puis composé au moment de l'exécution avec une FabriqueConcrete pour créer ses produits.
- Conséquences :
  - Comme le client est découplé des produits réels, nous pouvons substituer différentes fabriques pour obtenir différents comportements.

DP Abstract Factory : utilisations remarquables, patterns connexes

- Utilisations remarquables :
  - Très nombreuses : dans pattern d'architecture DAO pour construire instances d'objets d'accès aux données.
  - Toutes les méthodes de création qui retournent une interface ou une classe abstraite : Ex : javax.xml.parsers.DocumentBuilderFactory#newInstance()
- Patterns connexes :
  - Factory method : dans une fabrique abstraite, les méthodes pour créer les produits sont souvent implémentés à l'aide d'une fabrication (Factory Method).
  - Singleton : il ne devrait exister qu'une seule instance de chaque FabriqueConcrete.

Principes objets illustrés par ce patterns



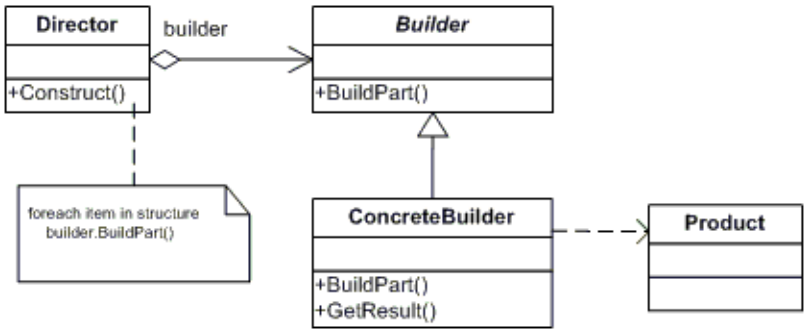


DP Builder permet de découpler construction d'objet complexe de sa représentation (1)

- Permet de construire des objets en plusieurs étapes selon un processus variable (à la différence des Fabriques) : l'algo qui crée l'objet est indépendant des parties qui le compose.
- Utile quand le client a besoin de construire des objets complexes ayant plusieurs implémentations ou représentations.

DP Builder : structure

http://www.dofactory.com/net/builder-design-pattern



DP Builder

Étude en autonomie du cours du Site developpez.com sur

le pattern Builder

http://rpouiller.developpez.com/tutoriel/java/design-patterns-gang-of-four

Le pattern d'architecture MVC

4IRC
Version 2011/2012 - Release 2016-2017

F. PERRIN

membre de UNIVERSITÉ DE LYON

Pattern MVC Modèle-Vue-Contrôleur : bibliographie

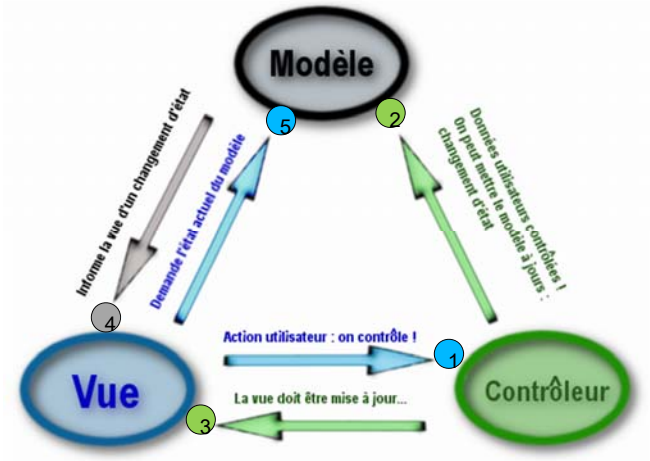
- L'étude de ce pattern se fera à partir de l'exemple du Site du Zéro :

  - http://fr.openclassrooms.com/informatique/cours/apprenez-a-programmer-en-java/mieux-structurer-son-code-le-pattern-mvc
  - D'autres liens bibliographiques sont intéressants :
  - http://fr.wikipedia.org/wiki/Mod%C3%A8le-Vue-Contr%C3%B4leur
  - http://code-weblog.com/developpement-dune-appli-android-basee-sur-un-mvc/

Pattern MVC : pourquoi ?

- Le pattern MVC est un pattern d'architecture.
- Il est recommandé de l'utiliser pour développer tout système interactif manipulant :
  - Des données (Modèle).
  - La présentation de ces données (Vue).
  - La logique de gestion des événements entre les données et la présentation (Contrôleur).
- Objectif : pouvoir modifier une couche sans impact sur les autres (Forte cohésion – Faible couplage) :
  - Changement de l'IHM (Vue) sans impact sur les objets métiers (Modèle).
  - Des objets métiers (Modèle) réutilisables, sans modification, dans plusieurs applications (gérées par contrôleurs différents).

Pattern MVC : schéma
- source siteduzero -



Pattern MVC : rôle de chaque entité

- La vue :
  - Elle donne une représentation du modèle (application graphique, page Web, etc.).
- Le modèle :
  - Il s'agit du cœur du programme qui gère les données.
- Le contrôleur :
  - Il reçoit les événements de la vue (action de l'utilisateur) et fait le lien avec le modèle.
  - Il contrôle la cohérences données.

Pattern MVC : principe de fonctionnement

L'utilisateur effectue une action à travers la Vue (Ex : clic sur un bouton).

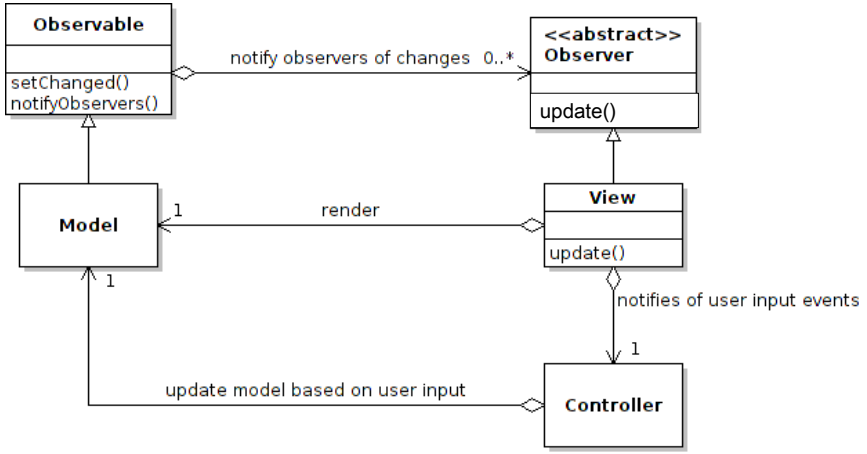
1. L'action est captée par le Contrôleur qui vérifie la cohérence des données et éventuellement les transforme afin que le modèle les comprenne.
2. Le Contrôleur envoie les données au Modèle et lui demande de se mettre à jour (Ex : variable qui change).
3. Contrôleur demande éventuellement à Vue de changer suite à l'action de l'utilisateur (pas suite à modification modèle…).
4. Modèle notifie à Vue qu'un changement nécessite qu'elle se mette à jour.
5. Informée du changement, la Vue s'actualise en allant chercher les données dans le Modèle (Ex : nouvelle valeur affichée).

Pattern MVC : DP GoF mis en œuvre

- Le pattern MVC repose essentiellement sur les patterns Observer, composite et Strategy.
- Le Modèle est le sujet observé.
  - La Vue observe ce sujet.
  - Le modèle notifie la Vue dès lors qu'il a un changement d'état suffisamment significatif pour en informer la Vue.
  - La Vue s'actualiser par un « pull » sur les données ou grâce à un « push » effectué par le modèle lors de la notification du changement.
- Le Contrôleur est la Strategy de la Vue :
  - Pattern Strategy permet d'encapsuler les morceaux de code qui changent.
  - Ainsi, des logiques de traitements différentes peuvent être associées à la même vue.
- La Vue :
  - Observe le modèle.
  - Est un composite : composant graphique (Jframe) composé de plusieurs composants graphiques.

Pattern MVC : structure

http://faculty.ycp.edu/~dhovemey/spring2012/cs320/lecture/lecture18.html



Conclusion

4IRC
Version 2011/2012 - Release 2016-2017

F. PERRIN

membre de UNIVERSITÉ DE LYON

Et finalement : que fait chaque Design Pattern ? (1)

- Patterns créateurs : ils définissent comment faire l'instanciation des objets et fournissent tous un moyen de découpler un client des objets qu'il a besoin d'instancier.
- Patterns structuraux : ils définissent comment organiser les classes d'un programme dans une structure plus large (séparant l'interface de l'implémentation).
- Patterns comportementaux : ils définissent comment organiser les objets pour que ceux-ci collaborent (distribution des responsabilités) et expliquent le fonctionnement des algorithmes impliqués.

Et finalement : que fait chaque Design Pattern ? (2)

– les patterns créateurs –

- Builder : permet de construire des objets en plusieurs étapes selon un processus variable (à la différence des Fabriques).
- Abstract Factory : permet à un client de créer des familles d'objets sans spécifier leurs classes concrètes.
- Factory Method : les sous-classes décident quelles sont les classes à créer.
- Prototype : permet de créer de nouvelles instances en copiant des instances existantes (clone() Java).
- Singleton : garantit qu'un objet et un seul est créé.

Et finalement : que fait chaque Design Pattern ? (3)
– les patterns structuraux –

- Adapter : enveloppe un objet et fournit une interface différente pour y accéder.
- Bridge : permet de faire varier l'implémentation *et* l'abstraction d'une hiérarchie de classes.
- Composite : les clients traitent les collections d'objets et les objets individuels de la même manière.
- Decorator : enveloppe un objet pour fournir un nouveau comportement.
- Facade : simplifie l'interface d'un ensemble de classes.
- Flyweight : facilite le partage d'un ensemble d'objets dont le grain est fin.
- Proxy : enveloppe un objet et en contrôle l'accès.

Et finalement : que fait chaque Design Pattern ? (4)
– les patterns comportementaux–

- Chain of Responsability : crée une chaine d'objet qui examinent une requête. Chaque objet considère la requête à son tour et la traite ou la transmet à l'objet suivant dans la chaîne.
- Command : encapsule une requête sous forme d'objet.
- Interpreter : fournit une représentation objet de la grammaire d'un langage afin d'évaluer des expressions.
- Iterator : fournit un moyen de parcourir une collection d'objet sans en exposer son implémentation.
- Mediator : simplifie la maintenance du système en centralisant la logique de contrôle sans que les éléments se connaissent mutuellement.

Et finalement : que fait chaque Design Pattern ? (5)
– les patterns comportementaux–

- Memento : sauvegarde et restaure l'état d'un objet (sérialisation java).
- Observer : permet de notifier des changements d'état à des objets.
- State : encapsule des comportement basés sur des états et utilise la délégation pour permuter ces comportements.
- Stategy : encapsule des comportements interchangeables et utilise la délégation pour décider lequel utiliser.
- Template Method : les sous-classes décident de la façon d'implémenter les étapes d'un algorithme.
- Visitor : ajoute des opérations à la structure d'un Composite sans modifier la structure elle-même.

Et finalement : comment effectuer une bonne conception ? (1)

Objectif :

Des programmes souples, extensibles et facile à maintenir.

Moyen :

Forte cohésion – Faible couplage.

Et finalement : comment effectuer une bonne conception ? (2)

- Efforcez-vous de coupler faiblement les objets qui interagissent.
- Encapsulez ce qui varie pour que sa modification n'ait pas d'impact sur le reste.
- Préférez la composition à l'héritage.
- Programmez des interfaces (au sens abstraction), non des implémentations (LSP : substitution de Liskov)
- Et usez de l'injection de dépendance (par constructeur, setter, fabrique couplée à fichier de configuration).
- Respectez le principe d'Hollywood : « Ne nous appelez pas, nous vous appellerons ».
- Limitez le nb de dépendance en respectant le principe « Ne parlez qu'à vos amis » (loi de Déméter).

Et finalement : comment effectuer une bonne conception ? (3)

- Gardez à l'esprit :
  - Une classe ne doit avoir qu'une seule raison de changer (SRP : Responsabilité Unique).
  - Une classe doit être ouverte à l'extension mais fermée à la modification (OCP : Ouverture – Fermeture).
  - Une classe doit dépendre d'abstractions et non pas de classes concrètes (DIP : Inversion de Dépendance).
  - Toute classe implémentant une interface doit implémenter chacune de ses fonctions (ISP : Ségrégation des Interfaces).

Et finalement : pourquoi utiliser les Design Patterns ?

- Concentrez-vous sur la conception, pas sur les patterns.
- Employez les patterns quand ils correspondent à un besoin naturel.
- Si une solution plus simple peut fonctionner, adoptez-la.
- Une seule situation impose de préférer un pattern a une solution simple : quand vous identifiez des points de variation.
- Si le changement est seulement hypothétique renoncez !

Conclusion (1)

Dialogue extrait de [1]

- « Maitre : Ta formation initiale est presque terminée, Petit scarabée. Quels sont tes projets ?
- Disciple D'abord, je vais aller à Disneyland ! Et puis je vais écrire des tas de programmes avec des patterns !
- M : Hé, attends une minute. Inutile de sortir l'artillerie lourde si tu n'en as pas besoin.
- D : Que voulez vous dire, Maître ? Maintenant que j'ai appris les design patterns, est-ce que je ne dois pas les utiliser pour obtenir le maximum de puissance, de souplesse et de maintenabilité ?
- M : Non. Les patterns sont un outil, et on ne doit utiliser un outil que lorsqu'on en a besoin. Tu as aussi passé beaucoup de temps à apprendre les principes de conception. Pars toujours de ces principes, et écris toujours le code le plus simple possible pourvu qu'il remplisse sa fonction. Mais si la nécessité d'un pattern se fait jour, emploie-le.

Conclusion (2)

Dialogue extrait de [1]

- D : Alors, je ne dois pas construire mes conceptions sur des patterns ?
- M : Cela ne doit pas être ton objectif quand tu entames une conception. Laisse les patterns émerger naturellement à mesure que celle-ci progresse.
- D : Si les patterns sont tellement géniaux, pourquoi tant de circonspection ?
- M : Les patterns peuvent introduire de la complexité et nous préférons toujours éviter la complexité lorsqu'elle est inutile. Mais les patterns sont puissants quand on les utilise à bon escient. Comme tu le sais, les patterns sont des solutions éprouvées issues de l'expérience et l'on peut les utiliser pour éviter des erreurs courantes. Ils constituent également un vocabulaire partagé qui nous permet de communiquer notre conception à d'autres développeurs.
- D : Mais comment sait-on qu'il est judicieux d'introduire un pattern?

Conclusion (3)

Dialogue extrait de [1]

- M : N'introduis un pattern que lorsque tu es sûr qu'il est nécessaire pour résoudre un problème dans ta conception ou pour faire face à un changement futur dans les exigences de ton application.
- D : Je crois que mon apprentissage va se poursuivre, même si je comprends déjà beaucoup de patterns.
- M : Oui, Scarabée. Apprendre à gérer la complexité et le changement dans le domaine logiciel est une poursuite sans fin. Mais maintenant que tu connais un bel assortiment de patterns, le temps est venu de les appliquer et de continuer à en apprendre d'autres.
- D : Attendez une minute, vous voulez dire que je ne les connais pas TOUS ?
- M : Scarabée… Tu as appris les patterns fondamentaux, mais tu vas découvrir qu'il en existe beaucoup plus, notamment des patterns qui s'appliquent uniquement à des domaines spécialisés tels que les systèmes concurrents et les systèmes d'entreprise. Mais maintenant que tu connais les bases, tu es prêt à les apprendre ! »

CONTACT

Domaine Scientifique de la Doua
43, bd du 11 Novembre 1918 - Bâtiment Hubert Curien
B.P. 2077 - 69616 Villeurbanne cedex - France

Tél. : (33) 04 72 43 17 00
Fax : (33) 04 72 43 16 84

www.cpe.fr

francoise.perrin@cpe.fr

membre de UNIVERSITÉ DE LYON

Et finalement…

- Programmez des interfaces et non des implémentations.
- Dépendez d'abstraction et non de classes concrètes.
- Découplez les clients des objets qu'ils créent/manipulent.
- Efforcez vous de coupler faiblement les objets qui interagissent.