Histoire des Design Patterns

Inspiré de reflexions issues de l'architecture (génie civil) dans les années 70.

Pattern (Christopher Alexander 1977)

Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

Model View Controller (MVC)

En 1979, Trygve Reenskaug, un des développeurs du langage Smalltalk invente le Modèle Vue Contrôleur (**??**).

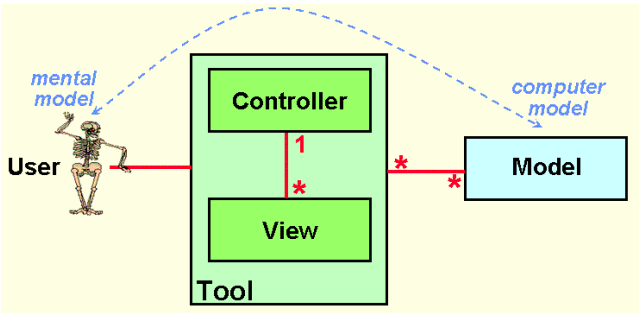


Image provenant de `http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html`

Histoire des Design Patterns (suite)

Le Gang of Four (GoF)

En 1995, le Gang of Four écrit le livre *Design Patterns : Elements of Reusable Object-Oriented Software* publié chez Addison Wesley.

- Le GoF : Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides.

General Responsability Assignement Software Patterns (GRASP)

Craig Larman écrit aussi en 1995 *Applying UML and Patterns : An Introduction to Object-Oriented Analysis and Design and Iterative Development*, publié chez Prentice Hall.

- Les modèles GRASP sont plus de bonnes pratiques que des modèles architecturaux.

Les 9 modèles GRASP

- Information Expert (8)
- Creator (10)
- Controller (11)
- Low Coupling (12)
- High Cohesion (14)
- Polymorphism (15)
- Pure Fabrication (16)
- Indirection (17)
- Protected Variations (18)

Contenu du livre Design Patterns : Elements of Reusable Object-Oriented Software

Catalogue de modèles de conception objet

Le catalogue est organisé est trois rubriques :

- les modèles de construction : comment créer les objets,
- les modèles de structuration : comment définir les relations entre les unités de programme,
- et les modèles de comportement : comment les objets communiquent.

Beaucoup d'autres modèles ont été conceptualisés.

Les 23 modèles du catalogue du GoF

Construction

- Abstract Factory (24)
- Builder (27)
- Factory Method (22)
- Prototype (**??**)
- Singleton (30)

Structuration

- Adapter (**??**)
- Bridge (**??**)
- Composite (**??**)
- Decorator (**??**)
- Façade (**??**)
- Flyweight (**??**)

Comportement

- Chain of Responsibility (**??**)
- Command (**??**)
- Interpreter (**??**)
- Iterator (**??**)
- Mediator (**??**)
- Memento (**??**)
- Observer (**??**)
- State (**??**)
- Strategy (**??**)
- Template Method (**??**)
- Visitor (**??**)

Information Expert

Expert en Information

La responsabilité doit être donnée à la classe qui dispose des informations pour l'exercer.

- respecte l'encaspuslation.

Encaspulation ou Information Hiding de David Parnas 1972

Encaspulation

Cacher les données, donc les champs en programmation objet. Ne les manipuler qu'à travers des méthodes. Ne rendre visible que ce qui est nécessaire.

- Les champs sont **private**.
- Seules les méthodes destinées à être utilisées en dehors de la classe sont **public**.

- améliore la robustesse du code.
- améliore l'évolutivité.

Creator

Créateur

Une classe A est responsable de la création d'un objet de la classe B si

- A agrège des objets de B ;
- A contient des objets de B ;
- A utilise des objets de B ;
- A possède les données nécessaires à l'initialisation des objets de B.

Controller

Contrôleur

Le contrôleur reçoit et coordonne les messages systèmes.

- Le contrôleur de façade gère l'accès à l'ensemble du système.
- Le contrôleur de scénario ou de session gère les événements contenus dans un même cas d'utilisation.

- Le contrôleur ne fait pas partie de l'interface utilisateur mais est le premier objet qui reçoit les événements qu'elle envoie.
- Le contrôleur n'effectue pas tous les traitements. Il délègue.
- Il correspond au contrôleur du MVC.

Low Coupling

Faible couplage

Définir les responsabilités en minimisant les liens entre les différents éléments pour faciliter la maintenance, l'ajout de fonctionnalités et la réutilisabilité.

- Les classes génériques doivent être faiblement couplées.
- Ne pas en abuser car le code peut devenir rapidement trop complexe.

Law of Demeter (Don't talk to strangers)

Loi de Déméter

Dans une méthode, l'envoi de message se limite à

- l'objet qui invoque la méthode **this** ;
- un paramètre de la méthode ;
- un champ de l'objet **this** ;
- un élément d'une collection qui est un champ de l'objet **this** ;
- un objet créé dans la méthode.

- Éviter de chaîner les méthodes en invoquant une méthode d'un objet retourné par une autre méthode :
  sale .getPayment().getAmount()
- La loi de Déméter aide à minimiser le couplage.

High Cohesion

Forte cohésion

Les éléments d'une unité fonctionnelle collaborent tous ensemble dans un périmètre bien délimité pour assurer un nombre limité de responsabilités.

- Il est difficile de trouver la bonne mesure pour assurer à la fois la forte cohésion et le faible couplage.

Polymorphism

Polymorphisme

Le polymorphisme consiste à utiliser le même nom pour réaliser des fonctions similaires. Le choix de la méthode est déterminé par le type de l'objet qui invoque la méthode.

- Éviter de tester dans le code avec des **if**, le type de l'objet qui invoque la méthode pour choisir la bonne méthode.

Pure Fabrication

Fabrication pure

Classe créée artificiellement pour réparer une architecture qui manque de forte cohésion et de faible couplage. Elle prend en charge l'échange des messages concernés et vise à les rétablir.

- encapsule une méthode qui ne trouve pas sa place ailleurs.
- Ne pas en abuser.

Indirection

Indirection

Classe qui sert d'intermédiaire entre deux classes pour assurer le faible couplage.

- par exemple, le contrôleur du MVC

Protected Variations

Variations protégées

Interface stable destinée à protéger des variations présentes dans le système actuel et dans ses évolutions.

- permet de respecter l'encapsulation.
- permet de respecter le principe ouvert fermé.

Simple Factory

- Isoler les parties du code contenant des **new** dans une méthode pour faciliter l'extensibilité, l'ajout de nouvelles classes.
- Ce n'est pas vraiment un design pattern.

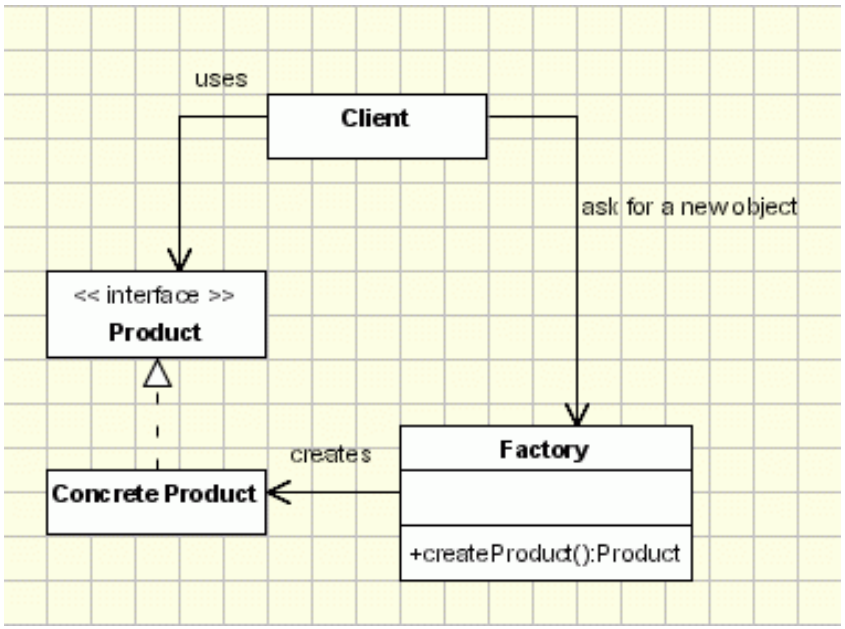


Image provenant de `https://hellosmallworld123.wordpress.com/2014/04/23/simple-factory-pattern-factory-methodand-abstract-factory-pattern/`

Simple Factory : exemple

```
public class IceCreamFactory {
  public IceCream getIceCream(String type) {
    IceCream cone = null;
    if (type.equals("Chocolate"))
      cone = new Chocolate();
    else if (type.equals("Strawberry"))
      cone = new Strawberry();
    else if (type.equals("Vanilla"))
      cone = new Vanilla();
    return cone;
  }
}

public class IceCreamShopV1 {
  private IceCreamFactory factory;
  public IceCreamShopV1(IceCreamFactory factory) {
    this.factory = factory;
  }
  public IceCream orderCone(String type) {
    IceCream cone;
    cone = factory.getIceCream(type);
    cone.scoop();
    return cone;
  }
  public static void main(String[] args) {
    IceCreamFactory factory = new IceCreamFactory();
    IceCreamShopV1 shop = new IceCreamShopV1(factory);
    shop.orderCone("Chocolate");
  }
}
```

Factory Method

Fabrication

Déférer la fabrication d'un objet à des classes dérivées.

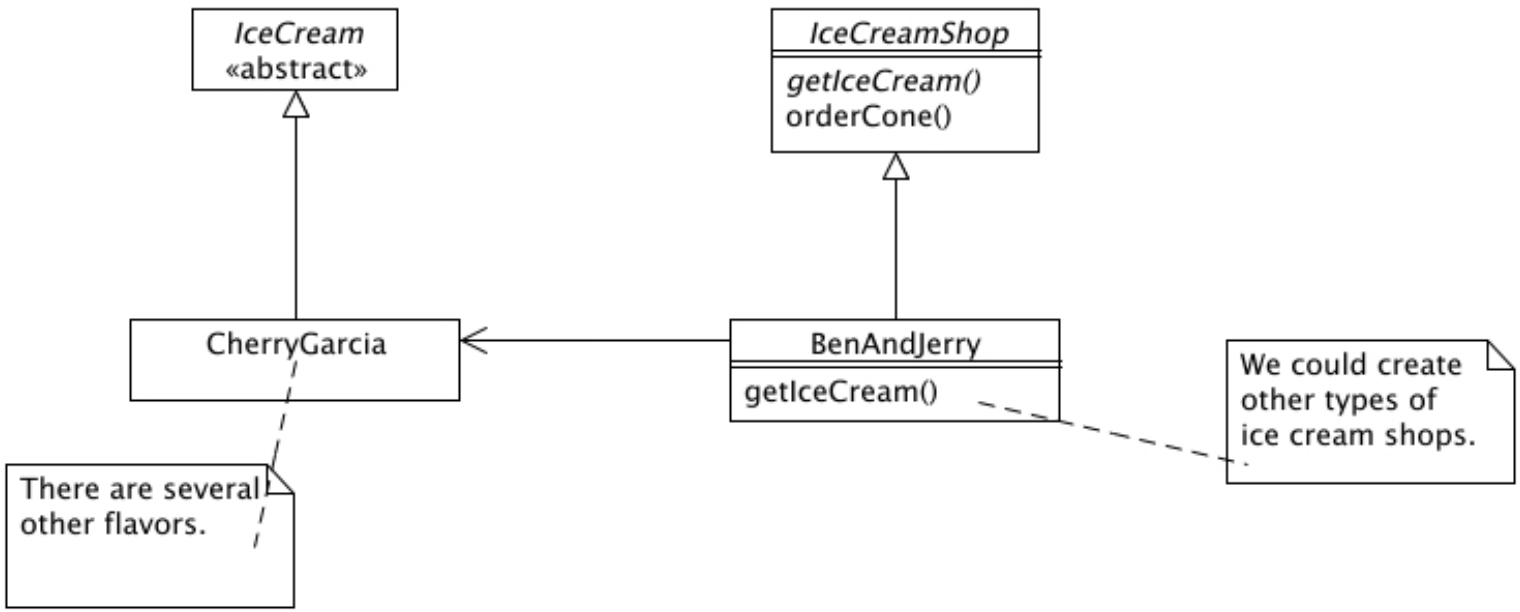


Image provenant de `http://www.people.westminstercollege.edu/faculty/ggagne/may2012/lab4/index.html`

Factory Method

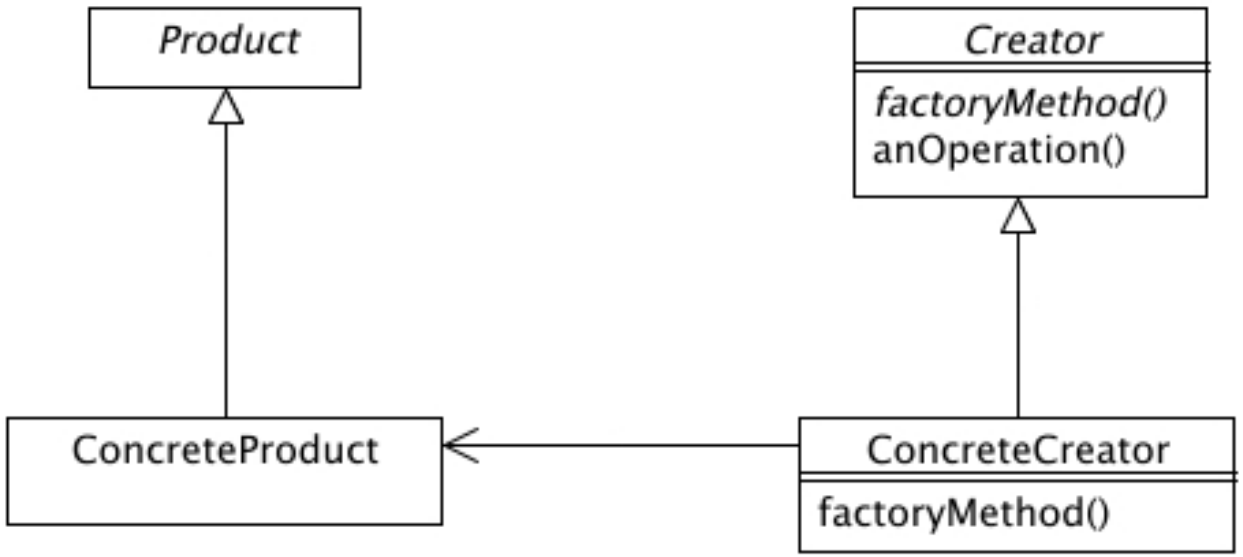


Image provenant de `http://www.people.westminstercollege.edu/faculty/ggagne/may2012/lab4/index.html`

Abstract Factory

Fabrication abstraite

Définir une interface pour créer des familles d'objets sans avoir à spécifier leurs classes concrètes.

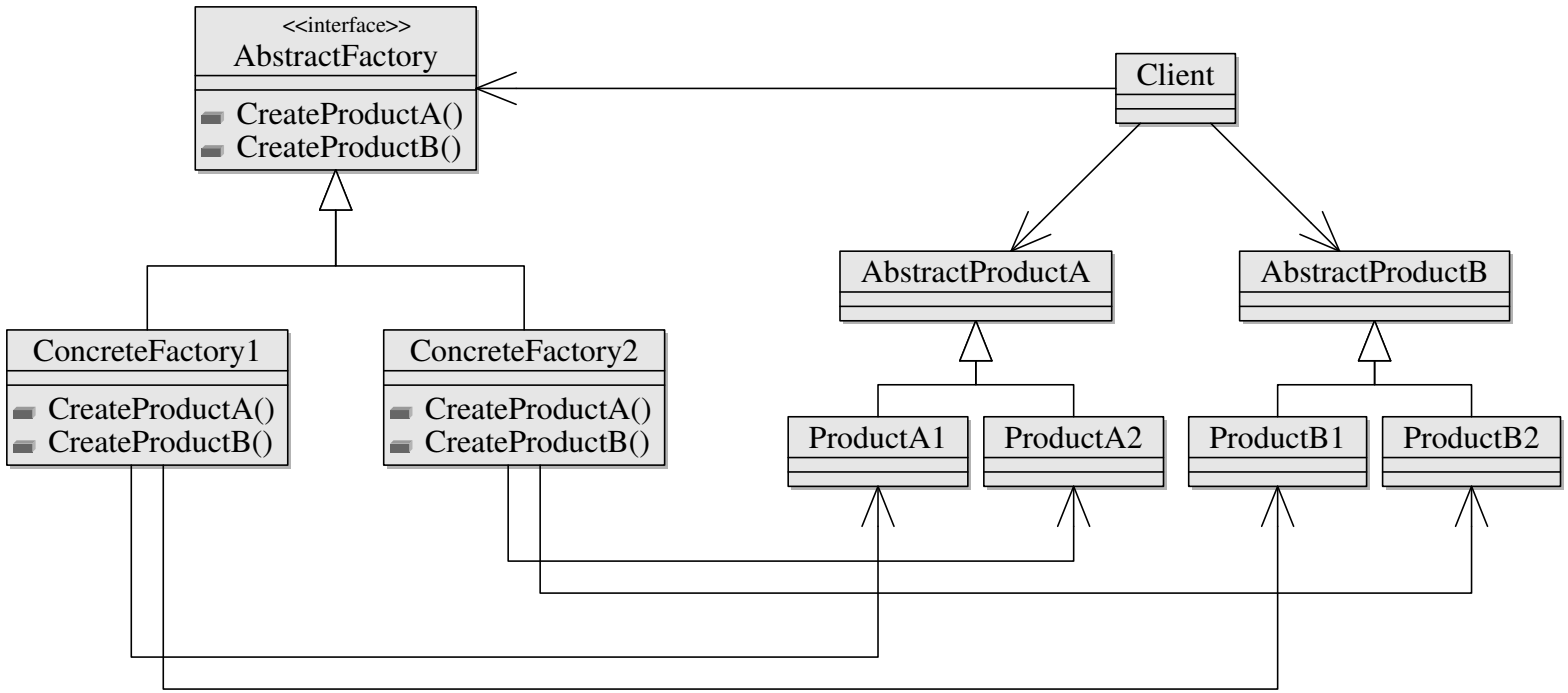


Image provenant de `http://thor.info.uaic.ro/~ogh/lop/lop.pdf`

Abstract Factory

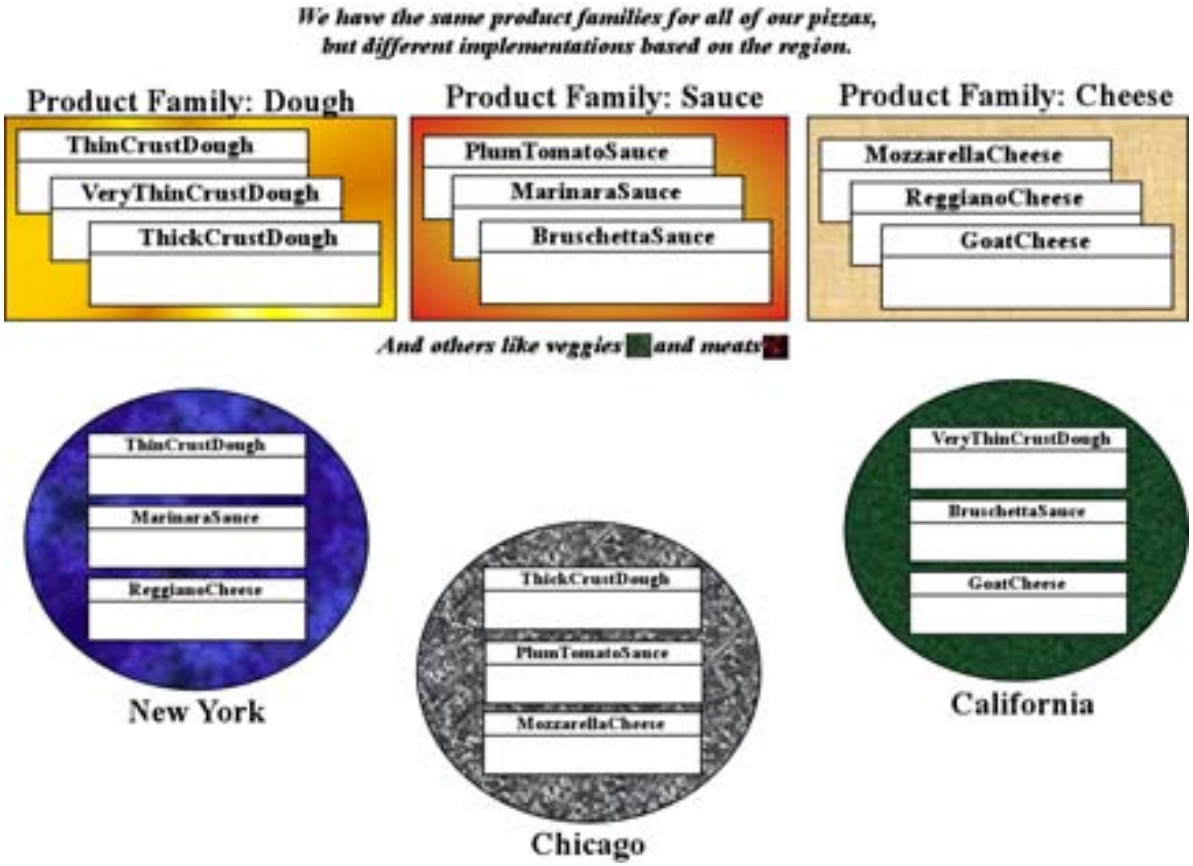


Image provenant de `http://www.cs.uah.edu/~rcoleman/CS307/DesignPatterns/DP05-AbstractFactory.html`

Abstract Factory

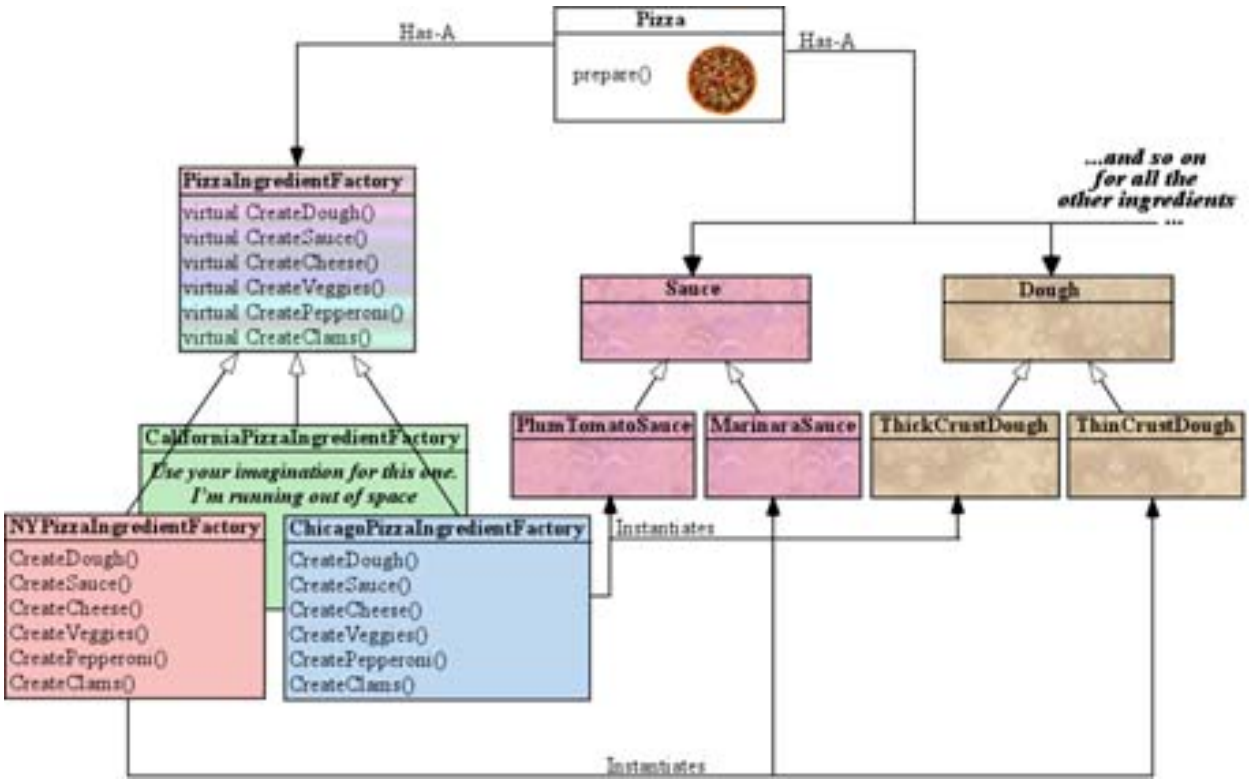


Image provenant de `http://www.cs.uah.edu/~rcoleman/CS307/DesignPatterns/DP05-AbstractFactory.html`

Builder

Monteur

Dissocier la construction de la représentation de l'objet. Un même procédé de construction engendre des représentations différentes.

- permet la construction d'un objet complexe par étape contrairement aux factories.
- L'implémentation de l'objet n'est pas visible et peut donc changer.

Builder

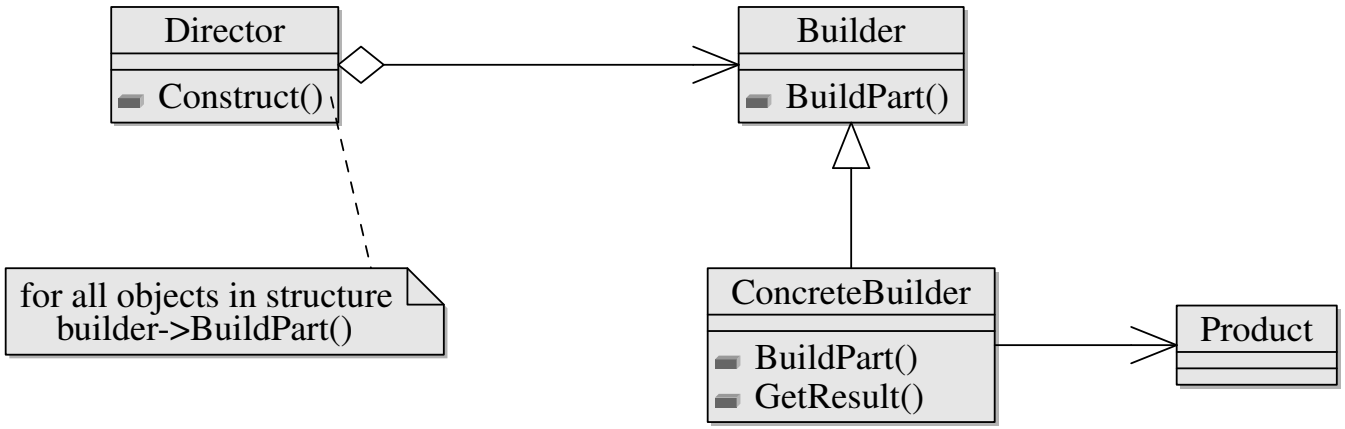


Image provenant de `http://thor.info.uaic.ro/~ogh/lop/lop.pdf`

- Le Product représente l'objet complexe que l'on construit.
- La classe abstraite Builder fournit une interface pour créer les parties de l'objet.
- Le ConcretBuilder construit et assemble l'objet Product en implémentant la classe Builder.
- Le Director construit l'objet complexe en utilisant l'interface Builder.

Builder : exemple

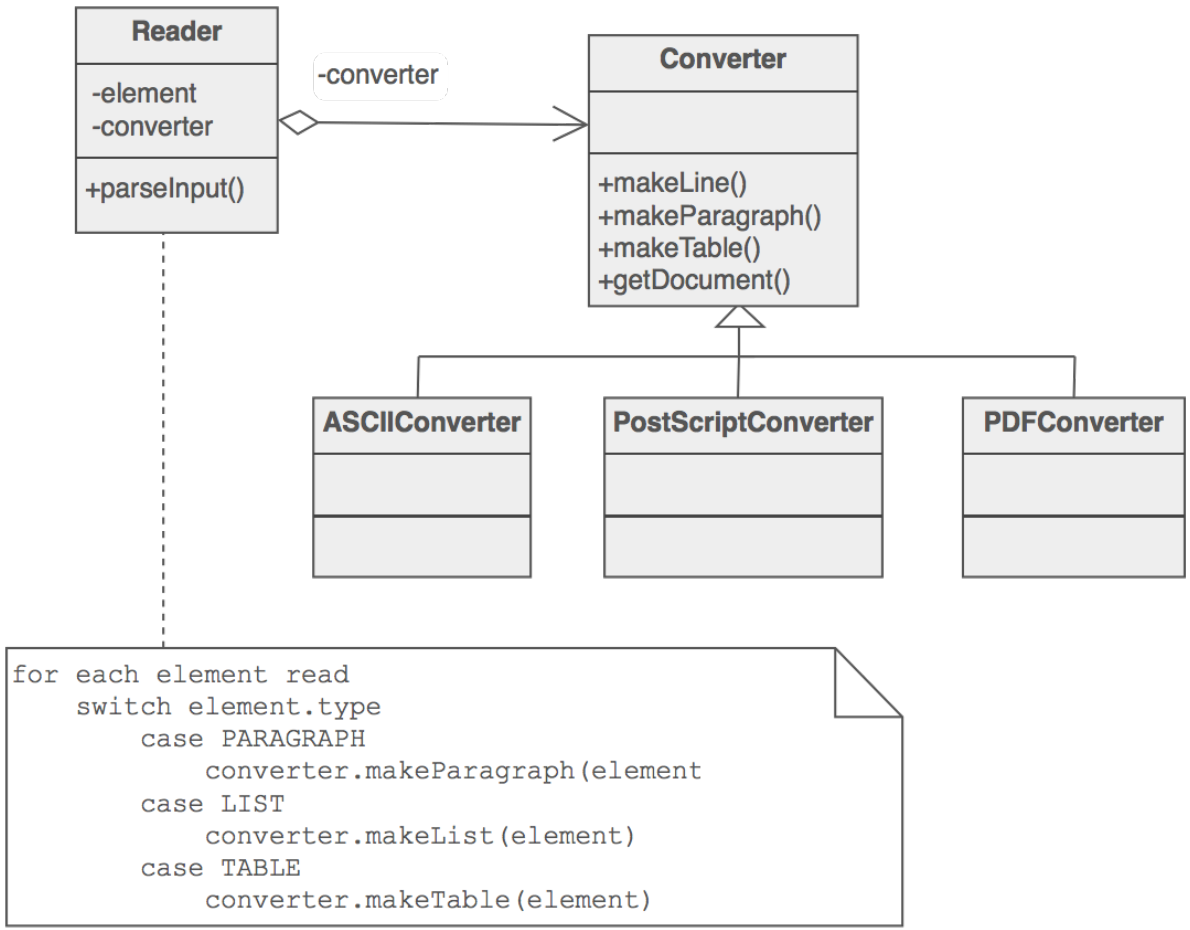


Image provenant de `https://sourcemaking.com/design_patterns/builder`

Singleton

Singleton

Garantir qu'une classe n'a qu'une instance et fournir un accès global à celle-ci.

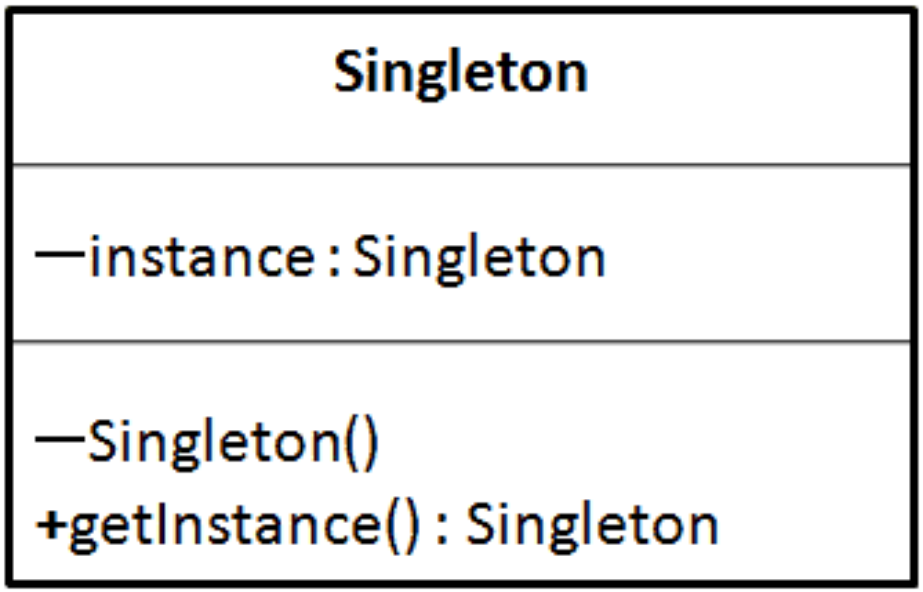


Image provenant de `http://java.boot.by/ocpjp7-upgrade/ch02.html`

Singleton : implémentation basique

```
public class Singleton {
  private static Singleton INSTANCE = null;

  private Singleton() {
  }

  public static Singleton getInstance() {
    if (INSTANCE == null) {
      INSTANCE = new Singleton();
    }
    return INSTANCE;
  }
}
```

- ne convient pas au programme multithread.

Singleton : implémentation thread-safe avec classe interne

```
public class Singleton {
  private Singleton() {
  }
  private static class SingletonHolder {
    private static final Singleton INSTANCE
        = new Singleton();
  }
  public static Singleton getInstance() {
    return SingletonHolder.INSTANCE;
  }
}
```

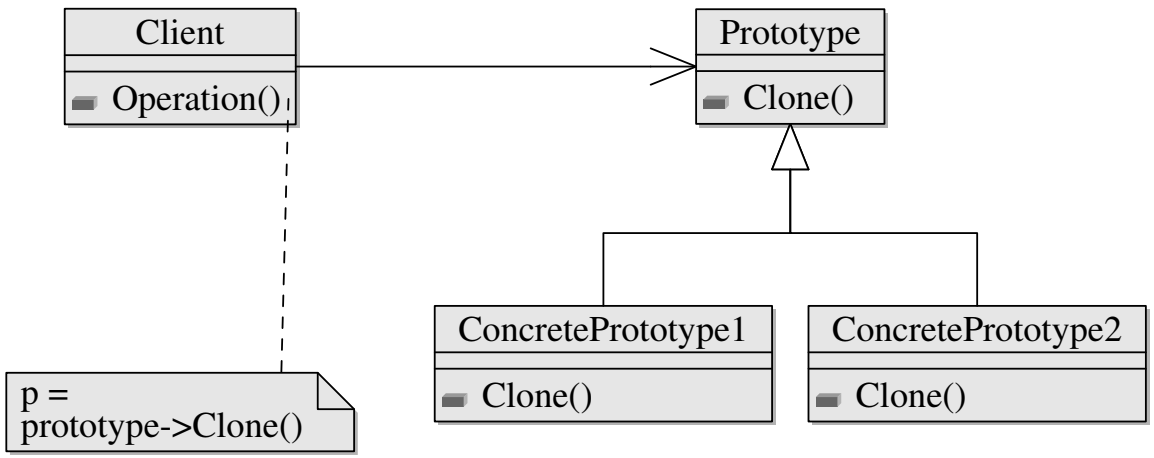
- ne fonctionne pas si le programme utilise plusieurs ClassLoaders.

Prototype

Prototype

Créer des objets par copie à partir d'un objet prototype.

Prototype


Client
Operation()
Prototype
Clone()
ConcretePrototype1
Clone()
ConcretePrototype2
Clone()
p =
prototype->Clone()

Decorator

Décorateur

Attacher des responsabilités supplémentaires à un objet de façon dynamique à l'exécution.

- Alternative à la spécialisation par héritage.
- respecte le principe Open/Close.
- Il est instancié avec une référence sur le composant décoré.
- La référence permet d'obtenir le comportement du composant.
- Le décorateur est appelé avec les mêmes méthodes que le composant décoré.
- Le comportement du composant est modifié dynamiquement à l'exécution.

Decorator

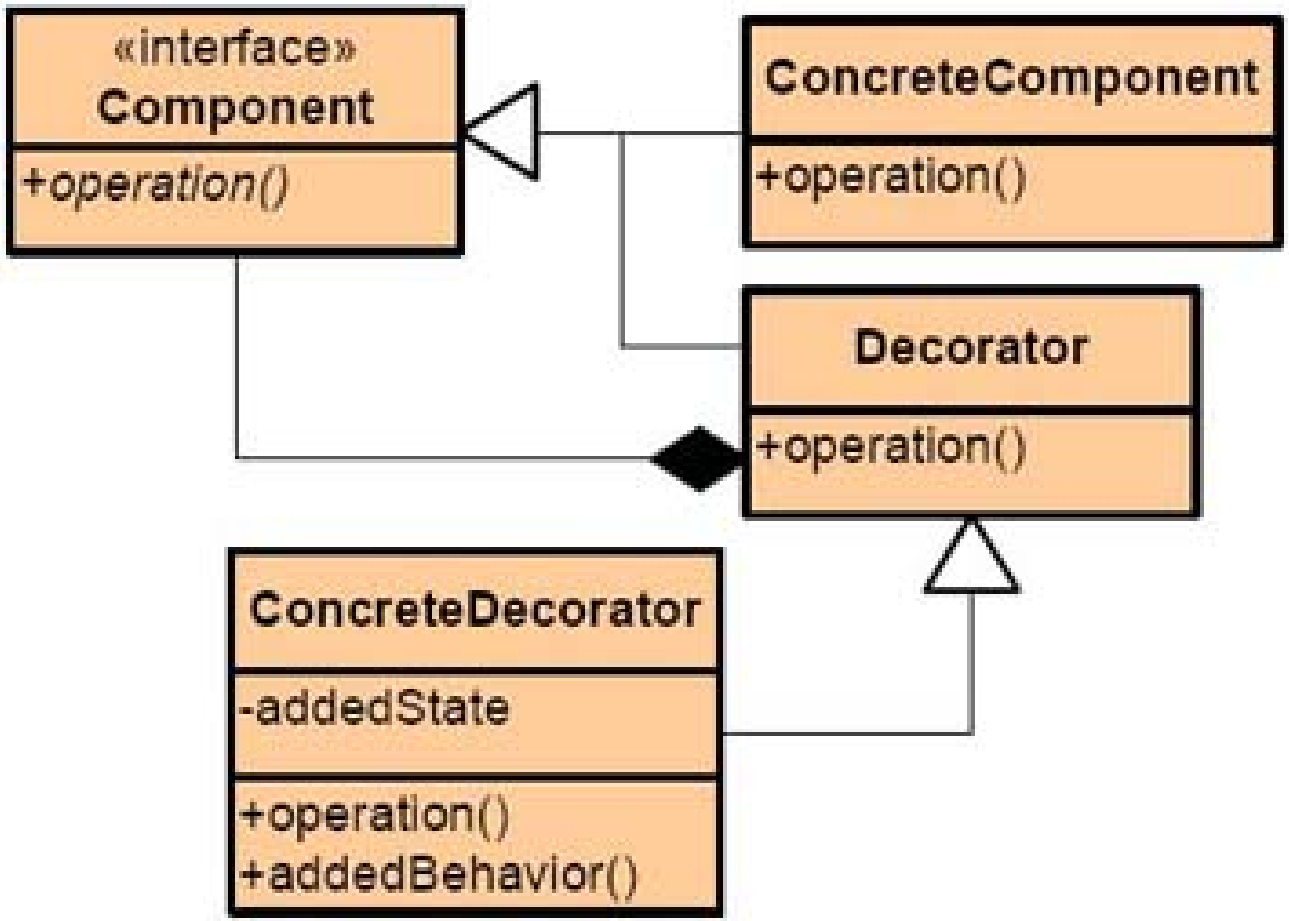


image provenant de `http://www.cs.uah.edu/~rcoleman/CS307/DesignPatterns/DP03-Decorator.html`

Decorator : exemple

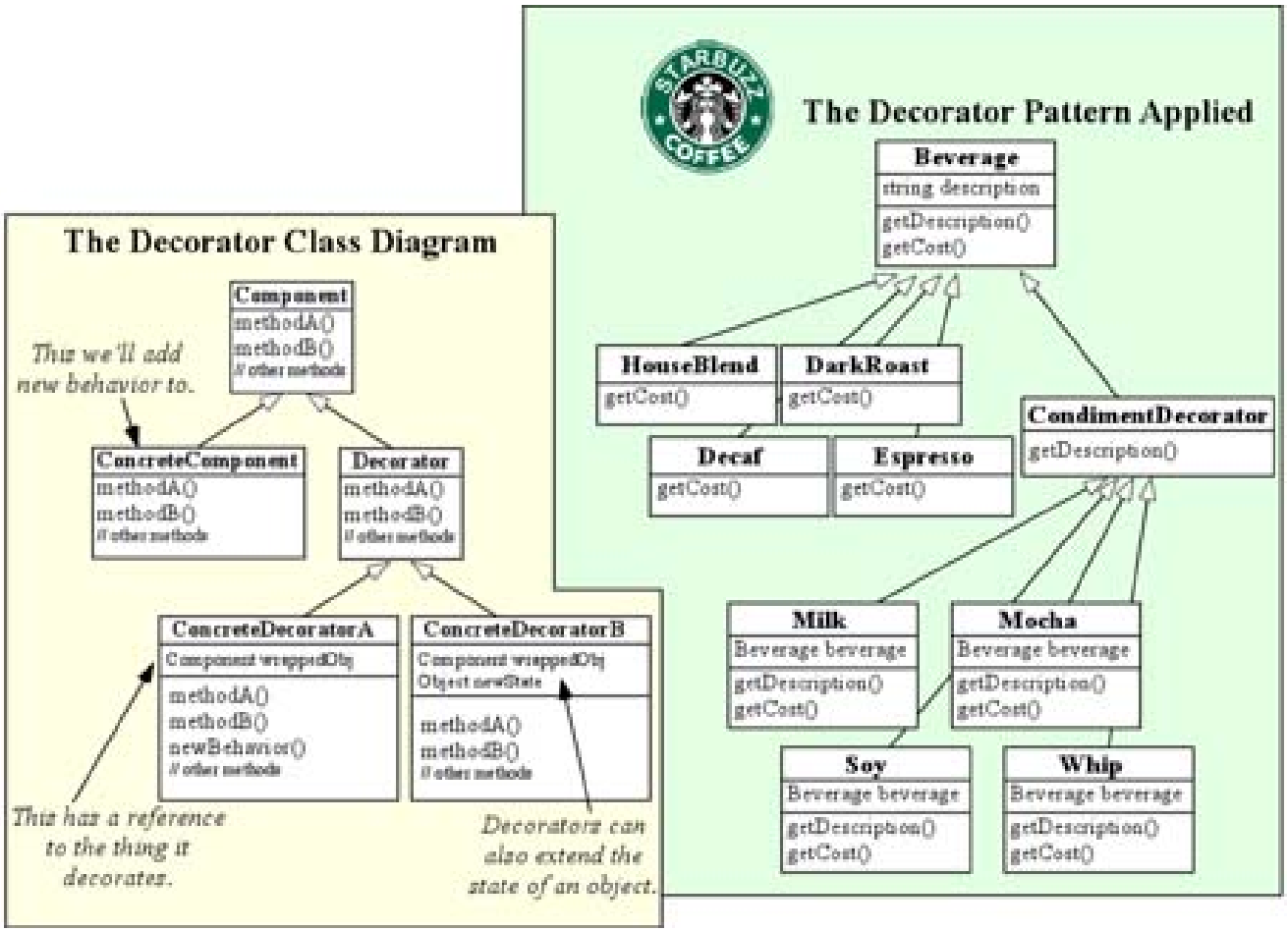


image provenant de `http://www.cs.uah.edu/~rcoleman/CS307/DesignPatterns/DP03-Decorator.html`

Decorator : java I/O

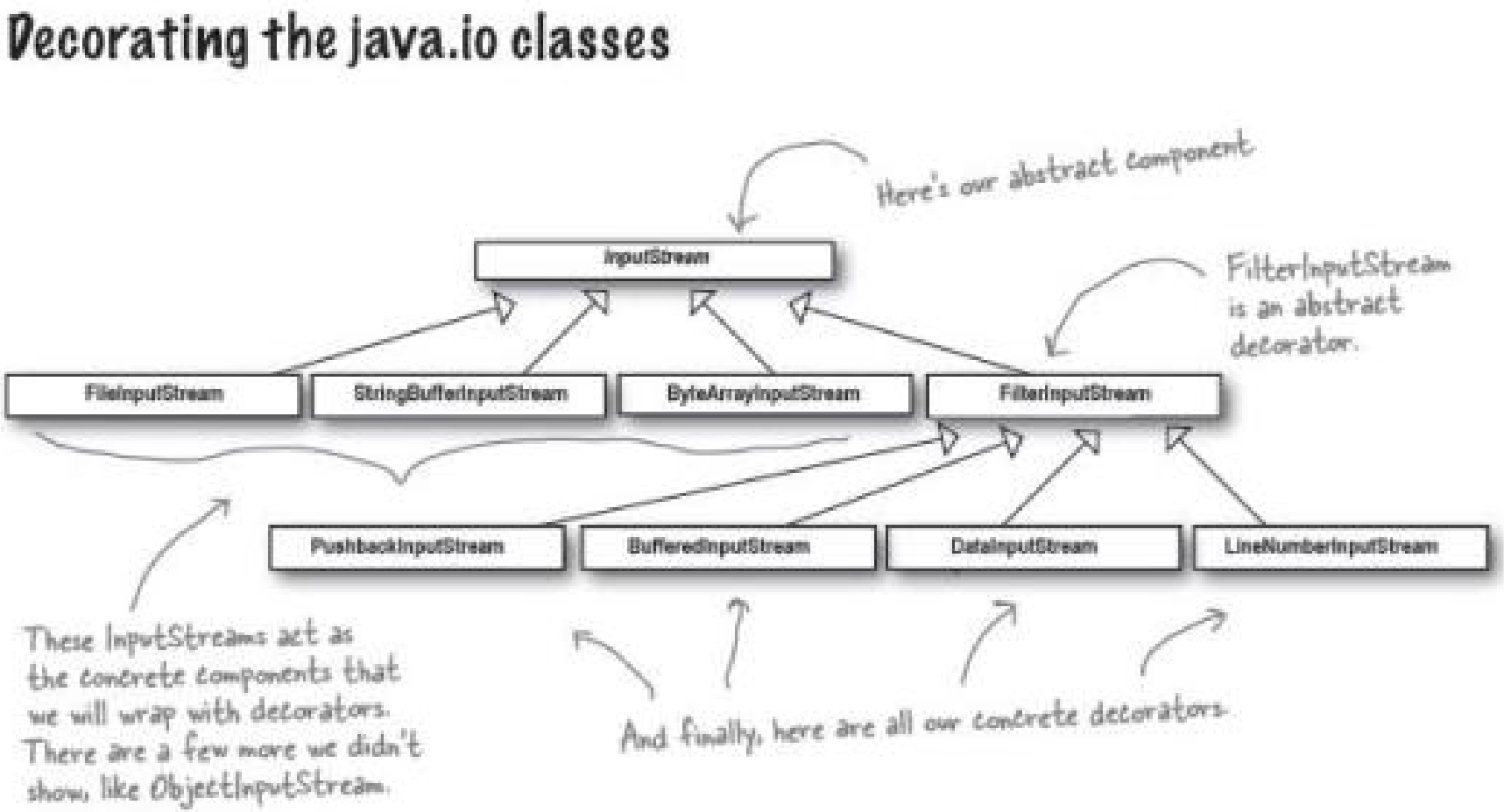


image provenant de Head First Design Pattens

Adapter

Adaptateur

Convertir l'interface d'une classe en une interface utilisable par un client.

- Rétrocompatibilité
- Object adapter : composition
- Class adapter : héritage multiple

Object adapter

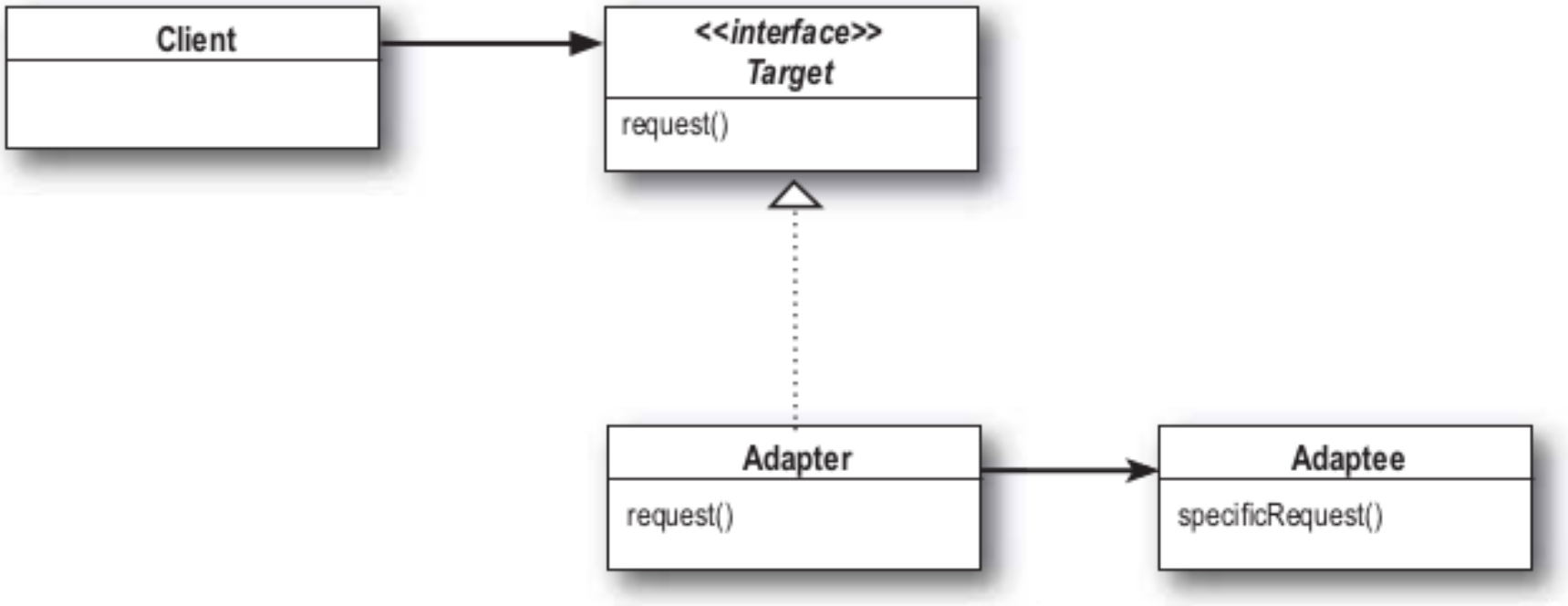


image provenant de Head First Design Pattens

Class adapter

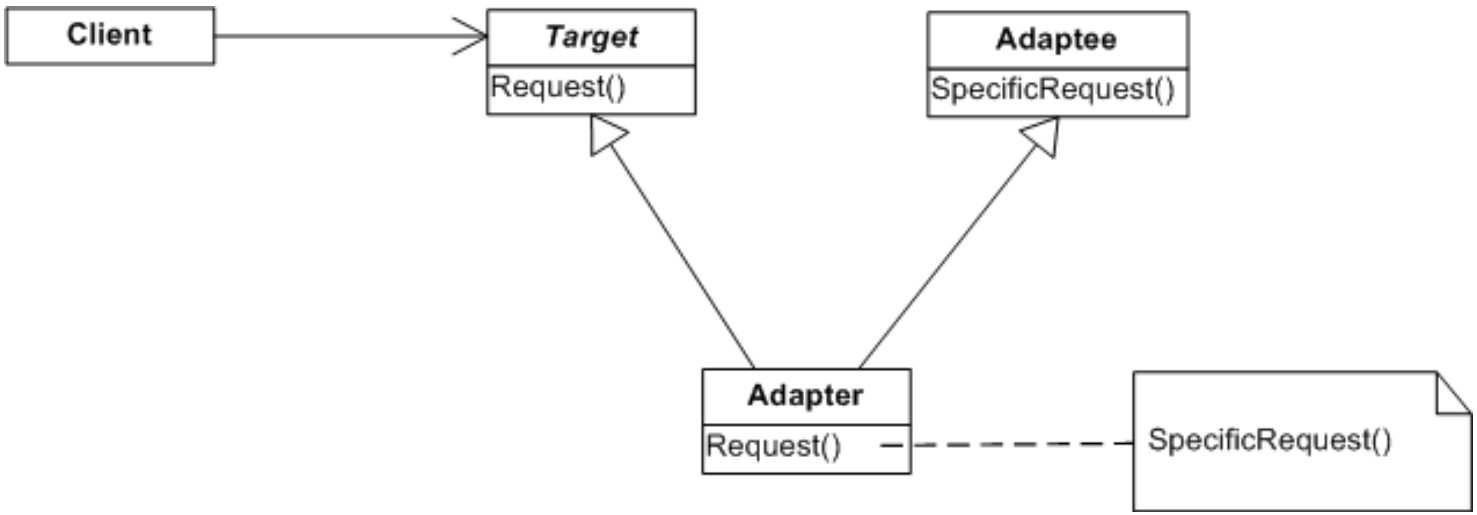


image provenant de `http://www.lepus.org.uk/ref/companion/Adapter(Class).xml`

Facade

Façade

Fournir une interface unifiée pour un ensemble d'interfaces d'un sous-système.

- Permet d'appliquer les bonnes pratiques suivantes :
  - Program to Interface Not Implementation.
  - Least Knowledge Principle (Loi de Déméter).

Facade

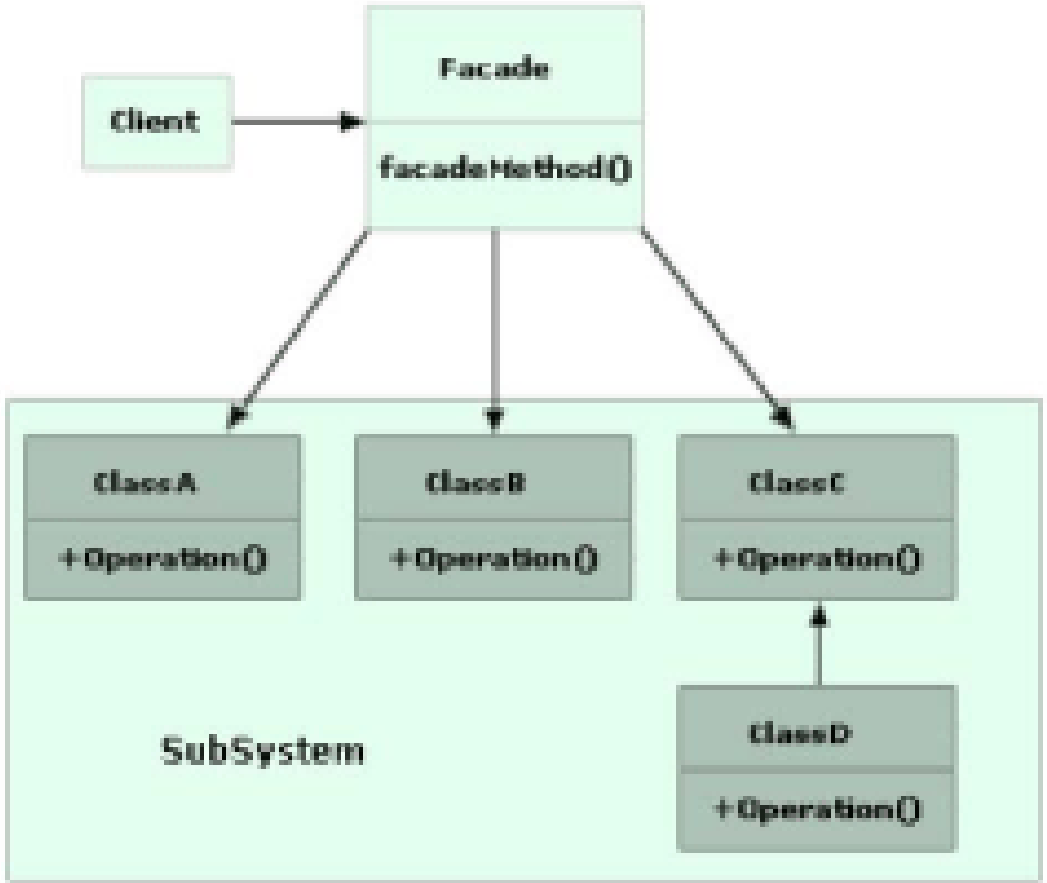


image provenant de `http://www.javajazzup.com/issue5/page39.shtml`

Facade : exemple 1

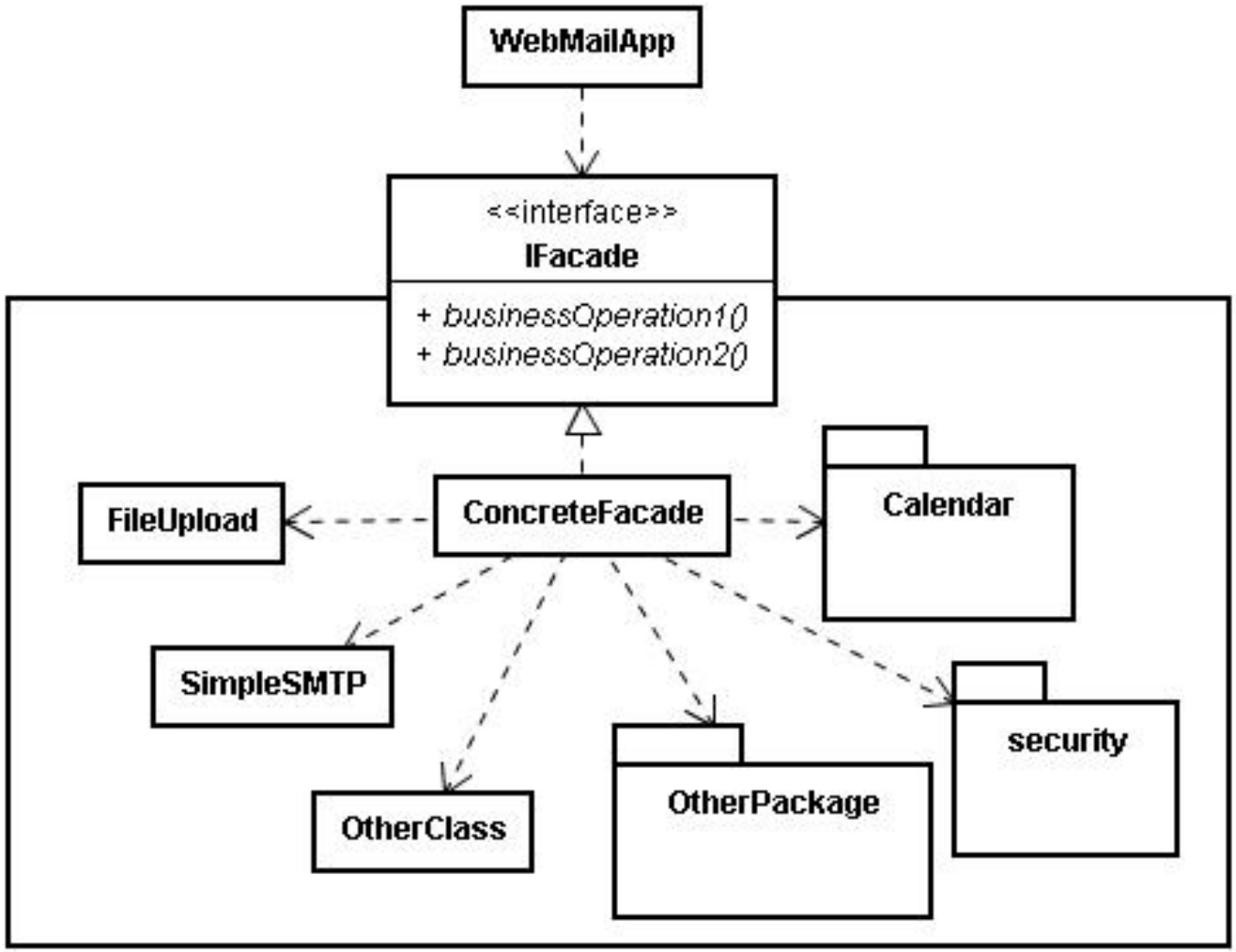


image provenant de `http://htepattern.net/design-pattern-facade/`

Facade : exemple 2

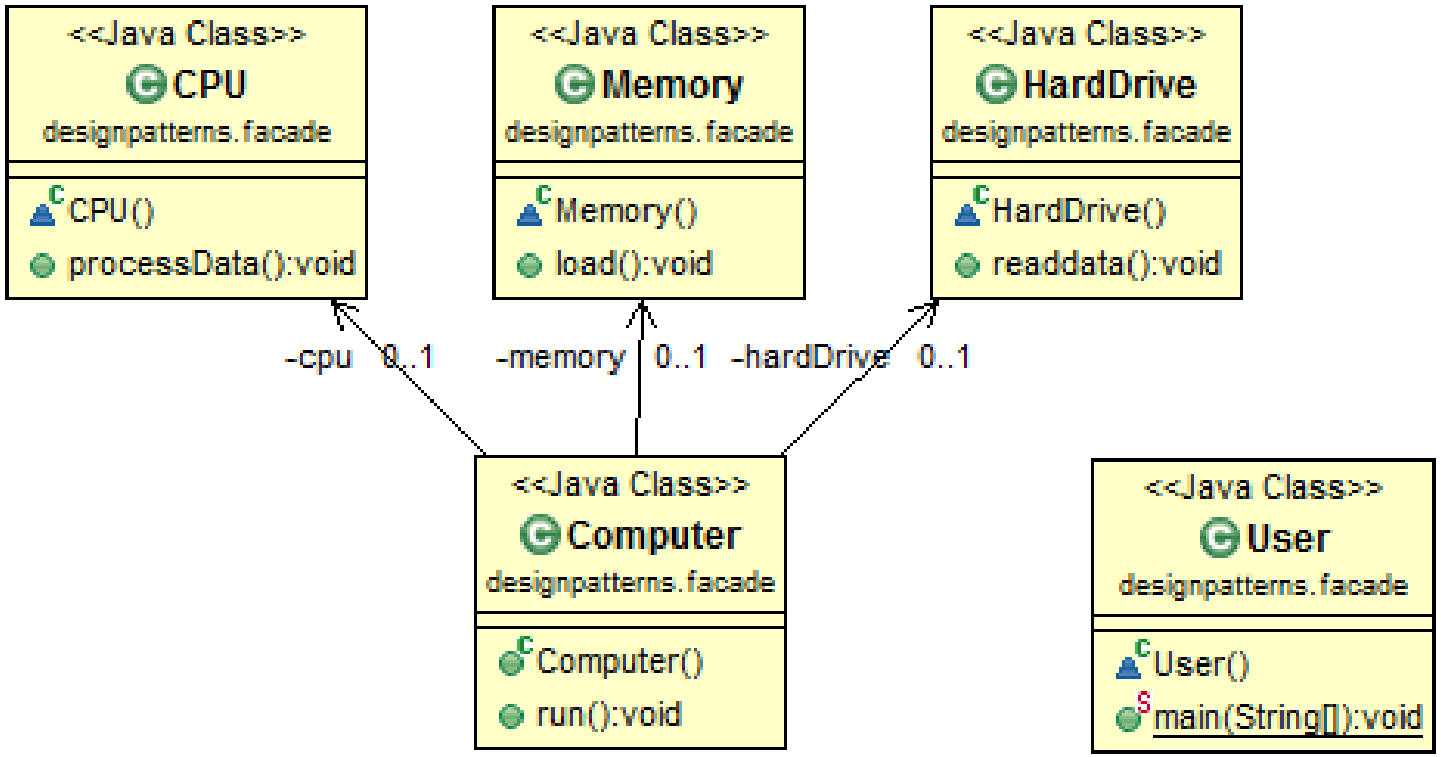


image provenant de `http://www.programcreek.com/2013/02/java-design-pattern-facade/`

Facade

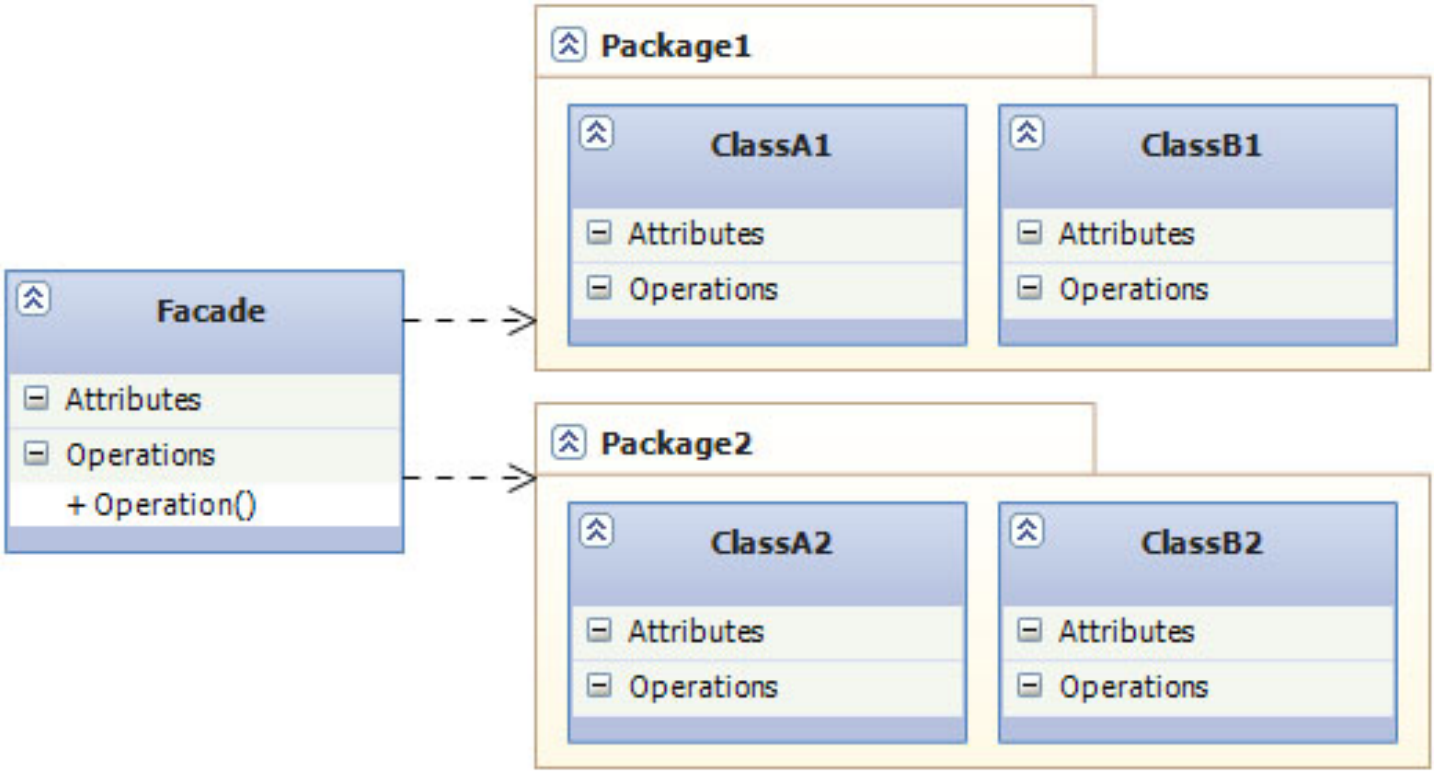


image provenant de

http://www.codeproject.com/Articles/438922/Design-Patterns-of-Structural-Design-Patterns

Composite

Composite

Organiser les objets en structure arborescente de manière à pouvoir utiliser un groupe d'objets comme s'il s'agissait d'un seul objet.

Par exemple,

- les répertoires et les fichiers ;
- les éléments d'une interface graphique ;
- un arbre syntaxique.

Composite

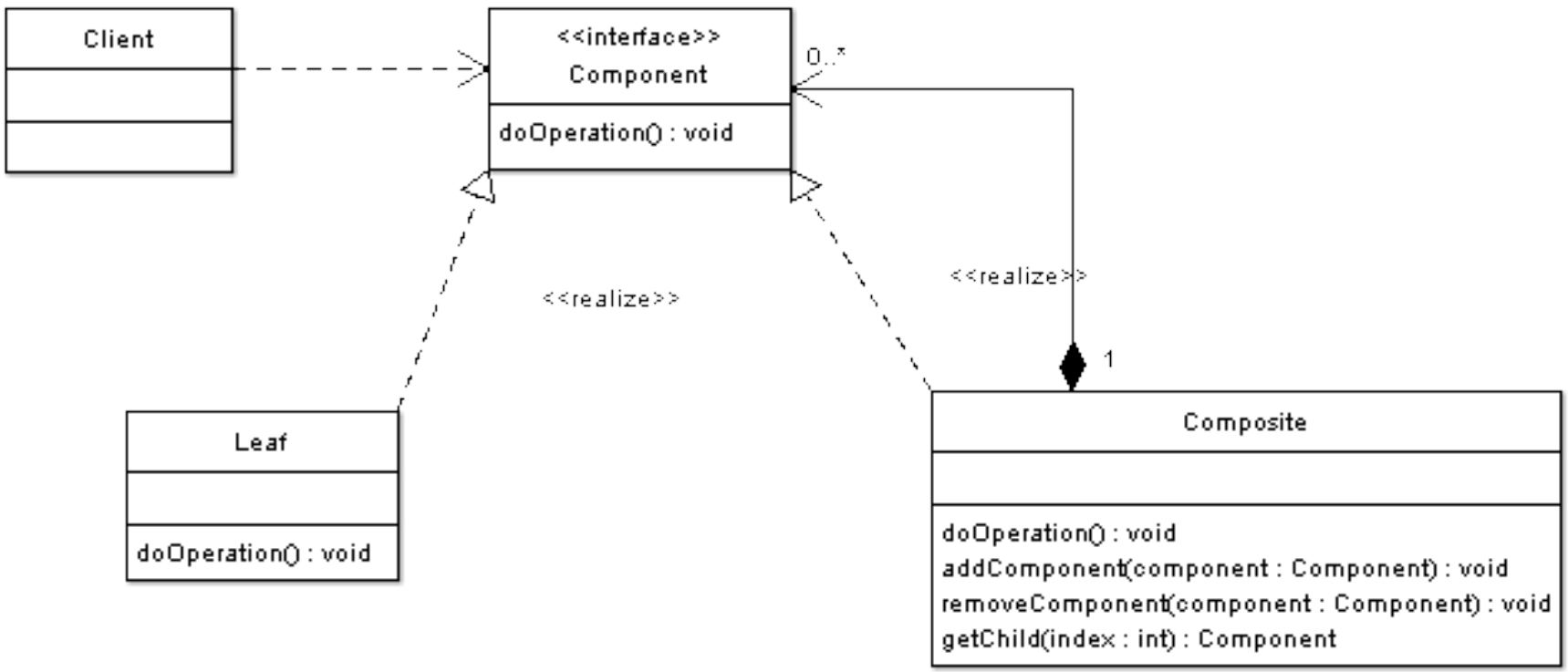


Image provenant de `http://www.oodesign.com/composite-pattern.html`

Composite : exemple

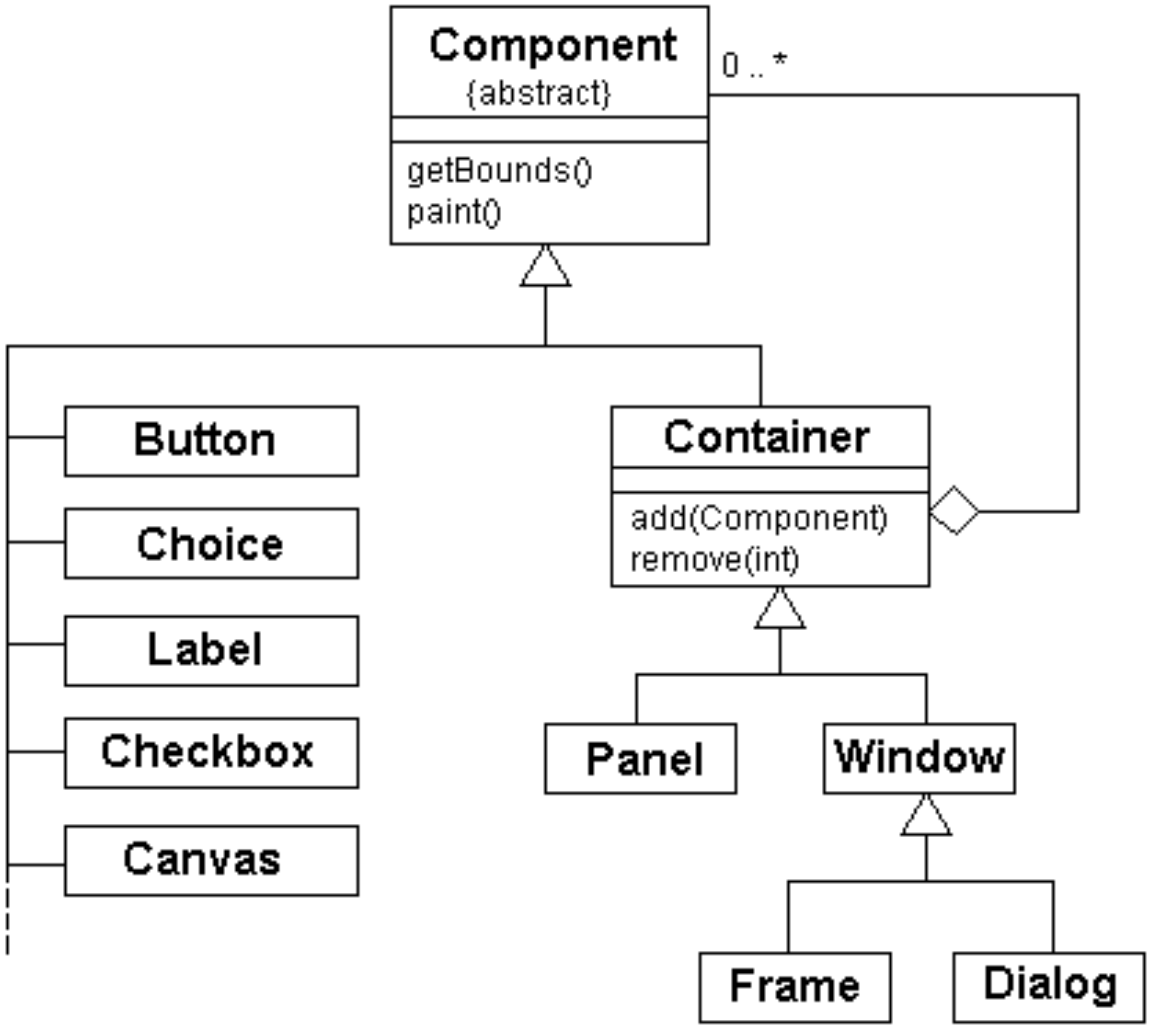


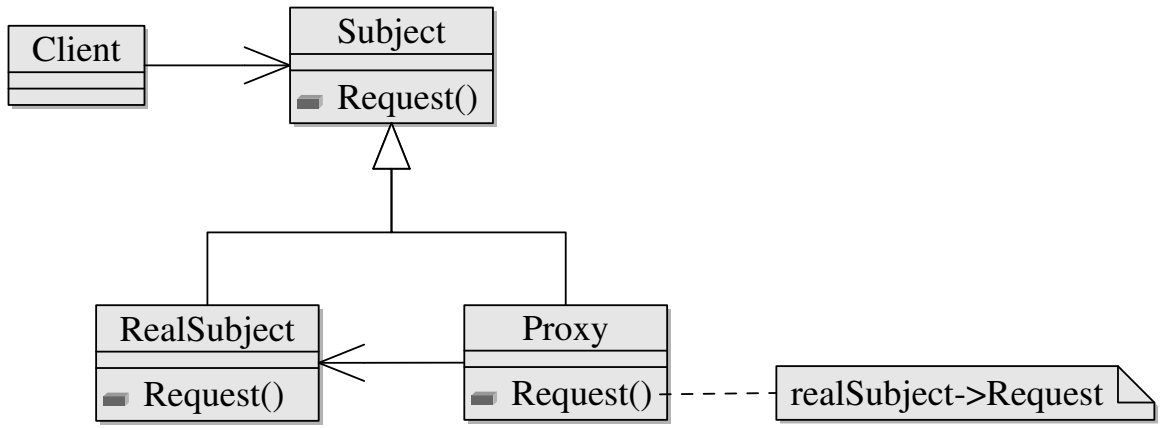
Image provenant de http://sourcemaking.com/uml/example-patterns/

Proxy

Procuration

Fournir une classe qui peut se substituer à une autre classe pour contrôler l'accès aux méthodes de cette dernière.

Proxy

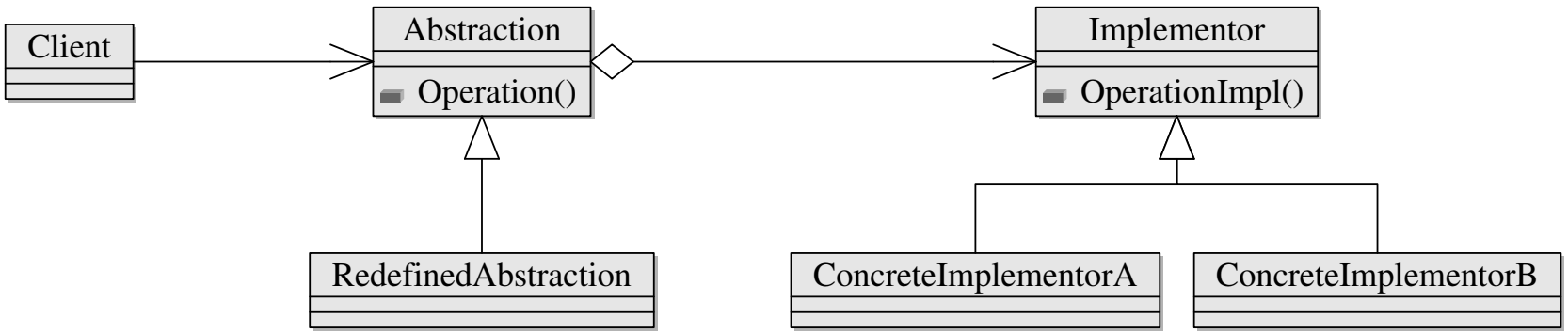

Client
Subject
Request()
RealSubject
Request()
Proxy
Request()
realSubject->Request

Bridge

Pont

Découpler une abstraction et son implémentation pour pouvoir les modifier indépendamment.

Bridge

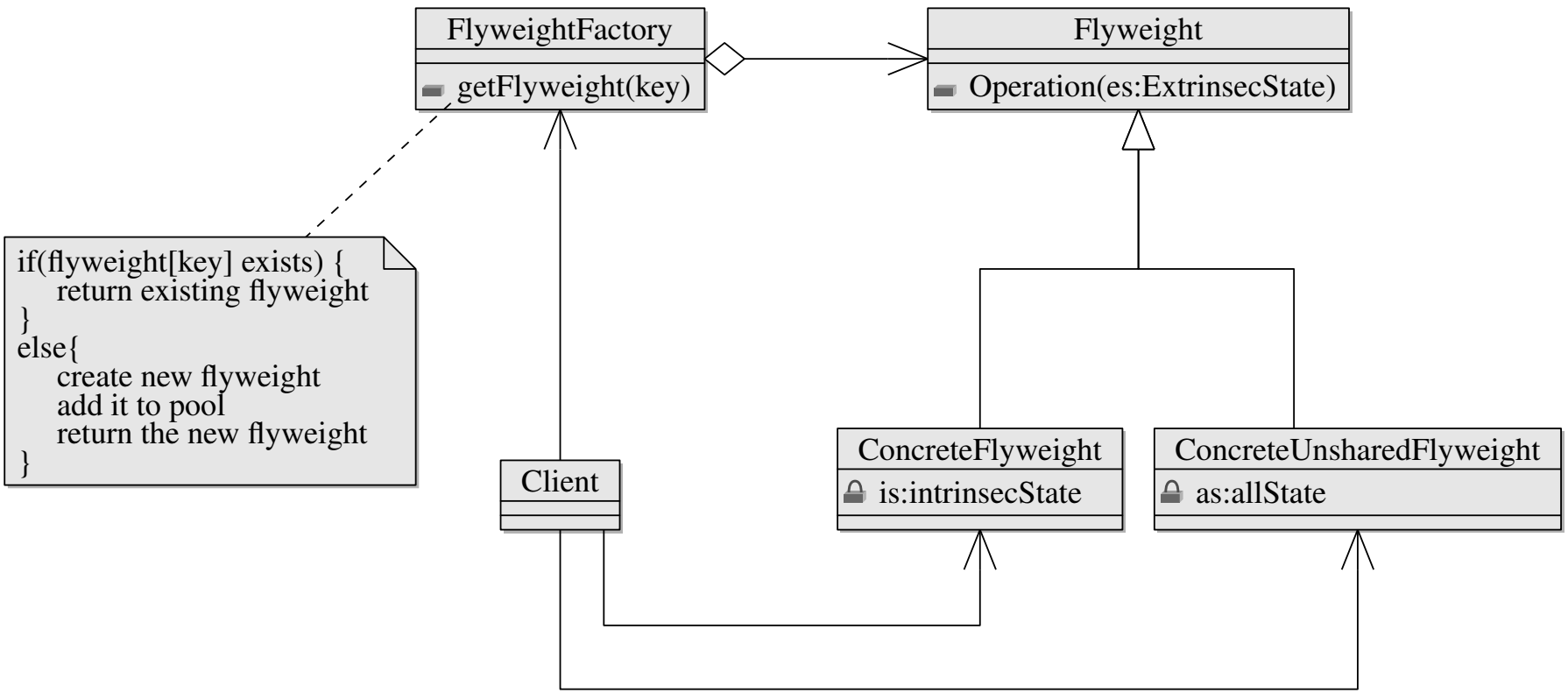

Client
Abstraction
Operation()
Implementor
OperationImpl()
RedefinedAbstraction
ConcreteImplementorA
ConcreteImplementorB

Flyweight

Poids mouche

Alléger l'occupation mémoire en factorisant les parties communes des objets et en les instanciant qu'une seule fois et gérer les parties qui diffèrent.

Flyweight


FlyweightFactory
getFlyweight(key)
Flyweight
Operation(es:ExtrinsecState)
if(flyweight[key] exists) {
return existing flyweight
}
else{
create new flyweight
add it to pool
return the new flyweight
}
Client
ConcreteFlyweight
is:intrinsecState
ConcreteUnsharedFlyweight
as:allState

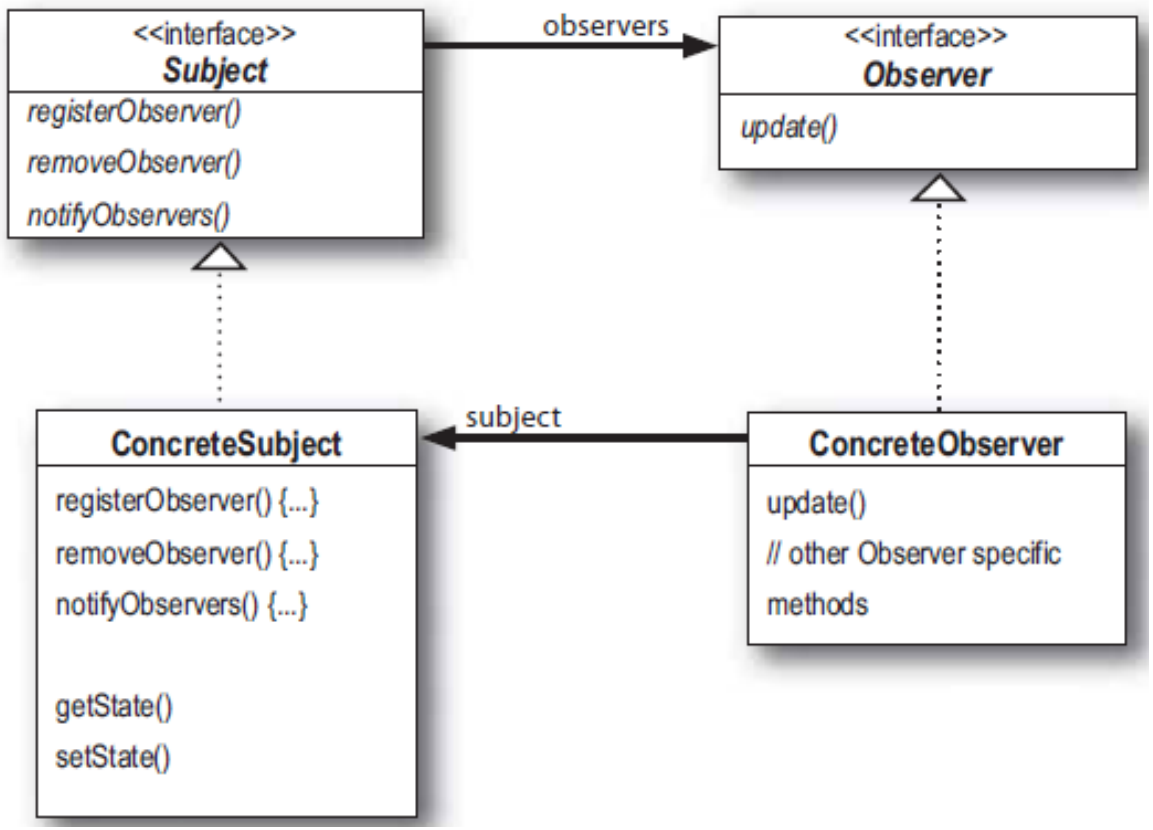
Observer

Observateur

Relation de type un à plusieurs : lorsqu'un sujet change d'état tous les objets qui l'observent sont avertis et mis à jour.

- Le sujet connaît ses observateurs. Il fournit une interface pour attacher et détacher les observateurs. Un nombre quelconque d'observateurs peut observer un sujet.
- L'observateur définit une interface de mise à jour pour les observateurs qui doivent être notifiés des changements dans un sujet.
- Le sujet concret mémorise les états pour les observateurs concrets. Il leur envoie une notification quand il change d'état.
- L'observateur concret gère une référence sur un sujet concret et implémente la mise à jour de l'observateur concret.

Observer



tiré du livre "Head First. Design Patterns"

- Objectif : typiquement réduire le couplage entre les composants d'une interface graphique et les données.

Observer : exemple

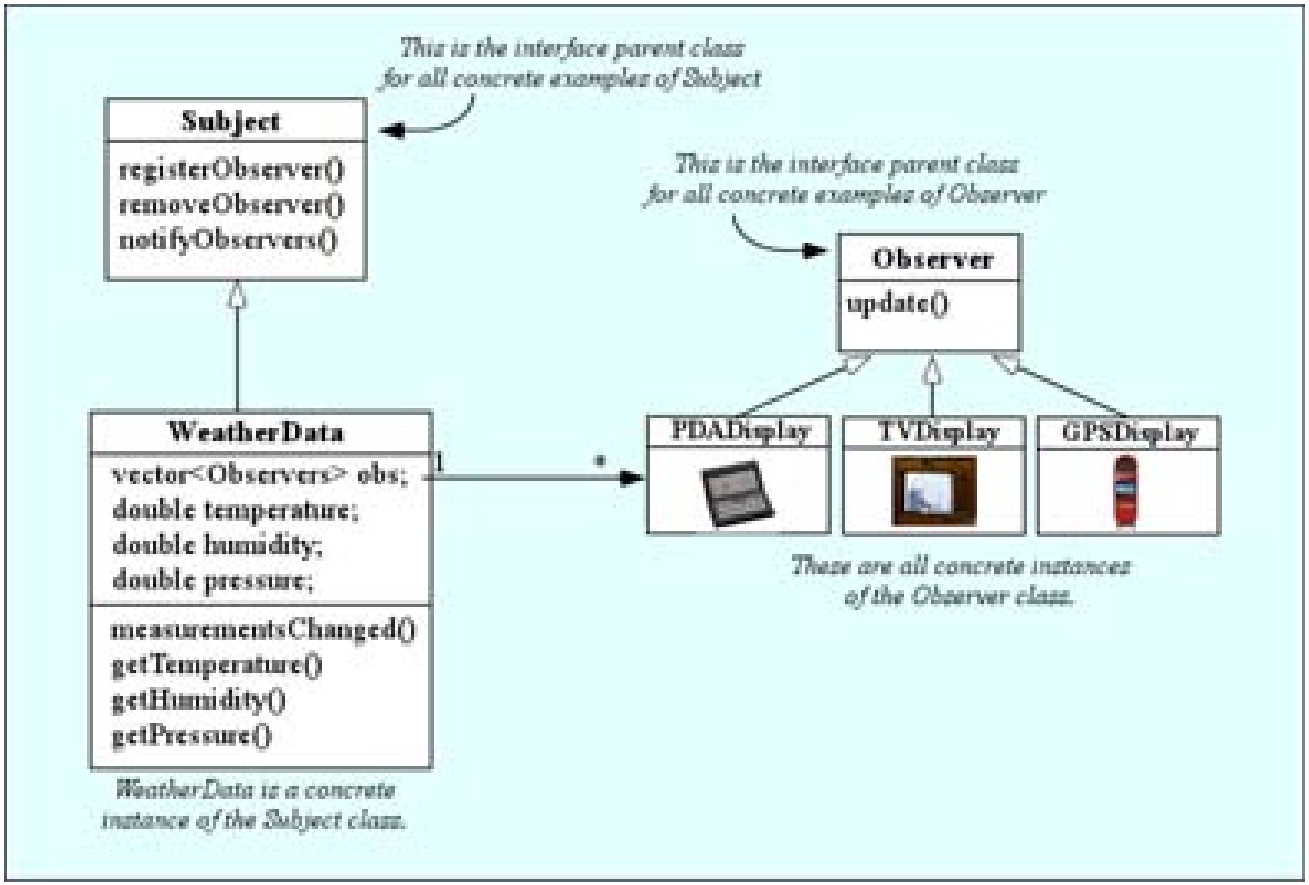


Image provenant de `http://www.cs.uah.edu/~rcoleman/CS307/DesignPatterns/DP02-Observer.htm`

Observer : mises à jour

- Le sujet concret informe les observateurs des mises à jour avec notifyObservers().
- On aurait pu confier à l'observateur la responsabilité de demander au sujet les notifications de mises à jour mais cette solution engendre souvent des erreurs en cas d'oubli.

Push

Le sujet envoie une information détaillée sur les modifications aux observateurs.

Pull

Le sujet envoie une information minimale sur les modifications et les observateurs réclament explicitement les détails dont ils ont besoin.

Observer : implémentation Java

java.util

- Interface Observer :
  - méthode **void** update(Observable o, Object arg)
- Classe Observable : il n'y a pas d'interface correspondant au sujet. Le sujet concret doit dériver de Observable.
  - Le sujet doit appeler la méthode **void** setChanged() ;
  - puis appeler
    - **void** notifyObservers () pour la méthode pull
    - ou **void** notifyObservers (Object arg) pour la méthode push. L'objet arg contient les informations envoyées aux observateurs.

- Le fait que Observable soit une classe et non une interface limite la réutilisabilité car la classe sujet doit étendre cette classe.

Interpreter

Interpréteur

Représenter la grammaire d'un langage, construire un arbre syntaxique et un interpréteur pour les expressions du langage.

- Le contexte peut servir par exemple à mémoriser des valeurs de variables.

Interpreter

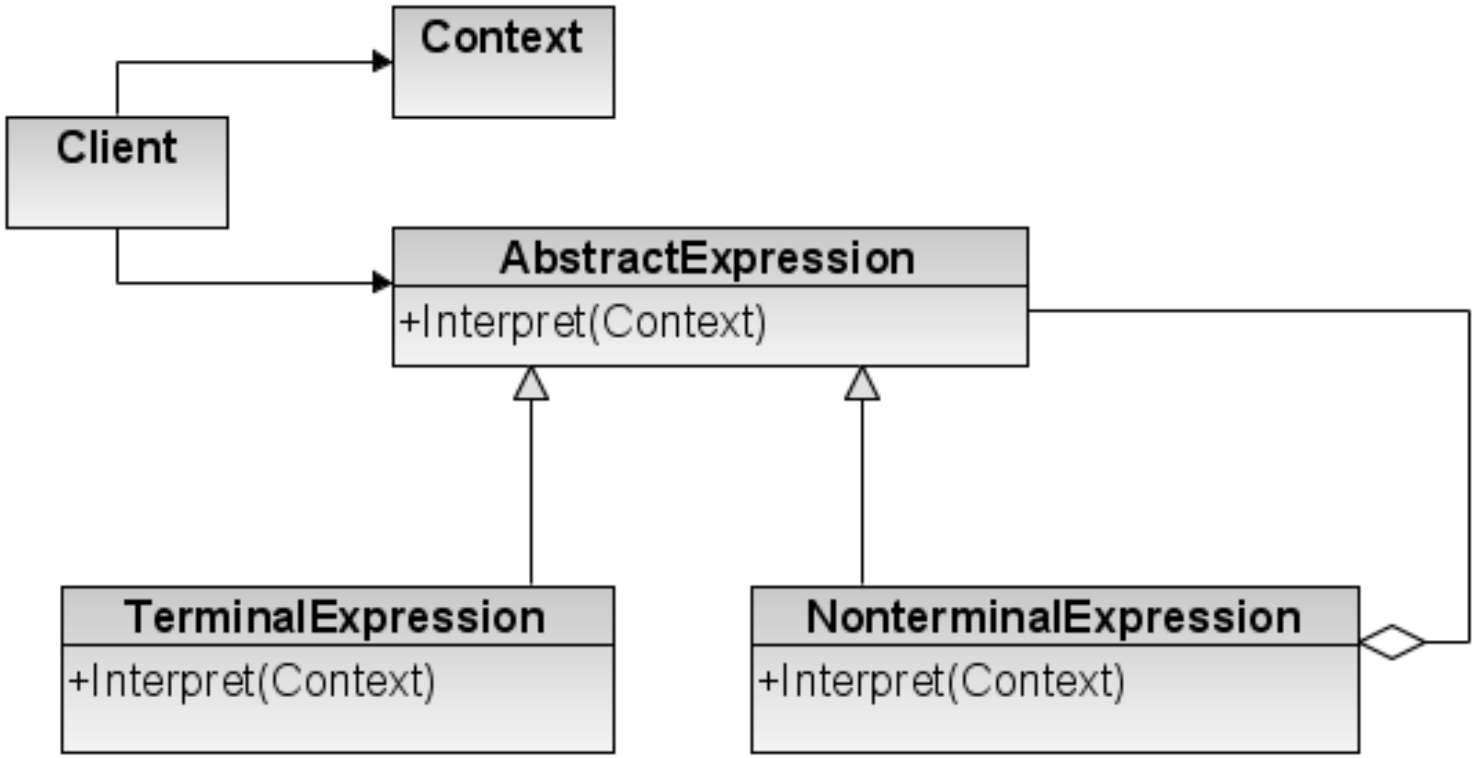


Image provenant de `https://javaobsession.wordpress.com/page/2/`

Visitor

Visiteur

Définir une nouvelle opération pour plusieurs classes :

- une collection d'objets de différents types,
- une arborescence d'objets suivant le pattern Composite.

- L'opération est externalisée.
- évite les **if** et les +instanceof+.

Visitor

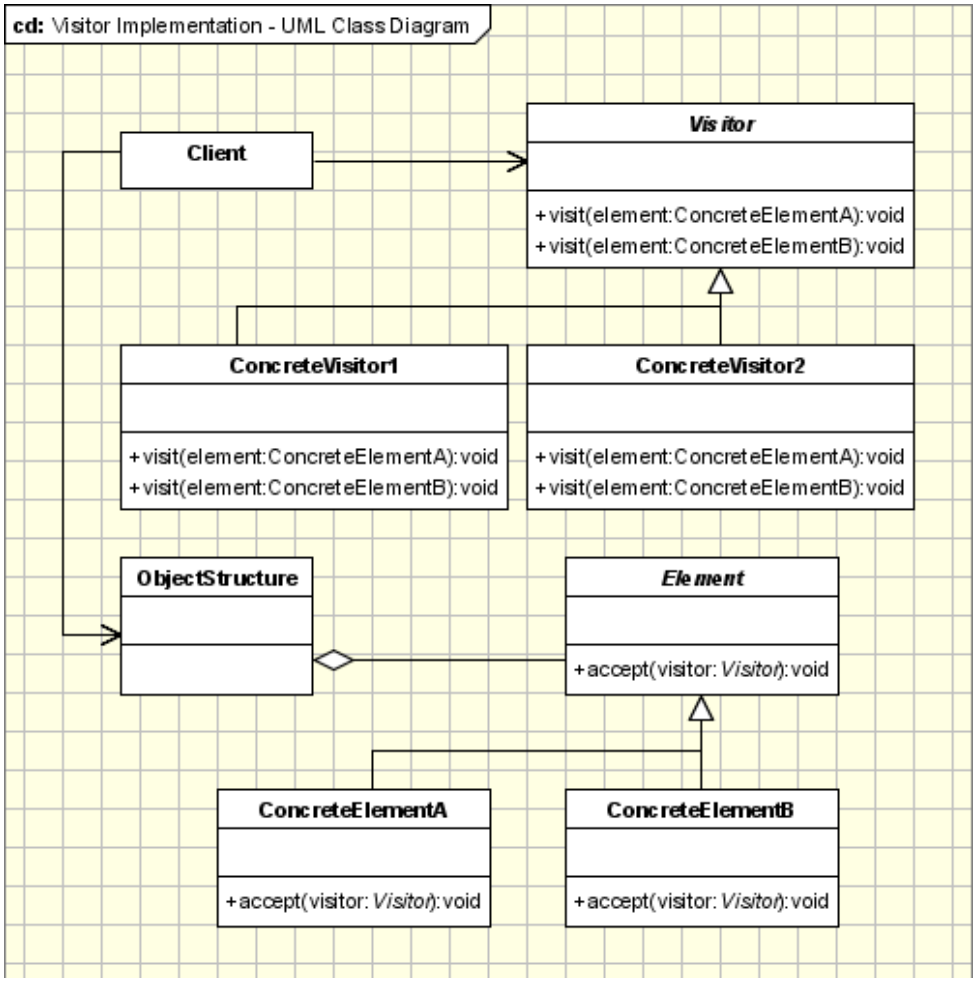


Image provenant de `http://www.oodesign.com/visitor-pattern.html`

Visitor : implémentation

Double dispatch

- On définit une interface Visitor avec les opérations visit(Element e) pour les différents types d'éléments à visiter.
- Les éléments concrets ont une méthode accept(Visitor v).

- Le visiteur a besoin de connaître les valeurs des champs des éléments visités. L'élément visité doit fournir des "getters" publics.

Template Method

Patron de méthode

Écrire un algorithme qui repose sur des traitements effectués dans des classes dérivées.

- Le squelette de l'algorithme est dans la méthode template.
- Certaines étapes sont définies ou redéfinies dans des classes dérivées.

- met en œuvre le principe d'Hollywwod
  - Don't call us, we'll call you.
  - Les composants de haut niveau appellent les composants de bas niveau.

Template Method

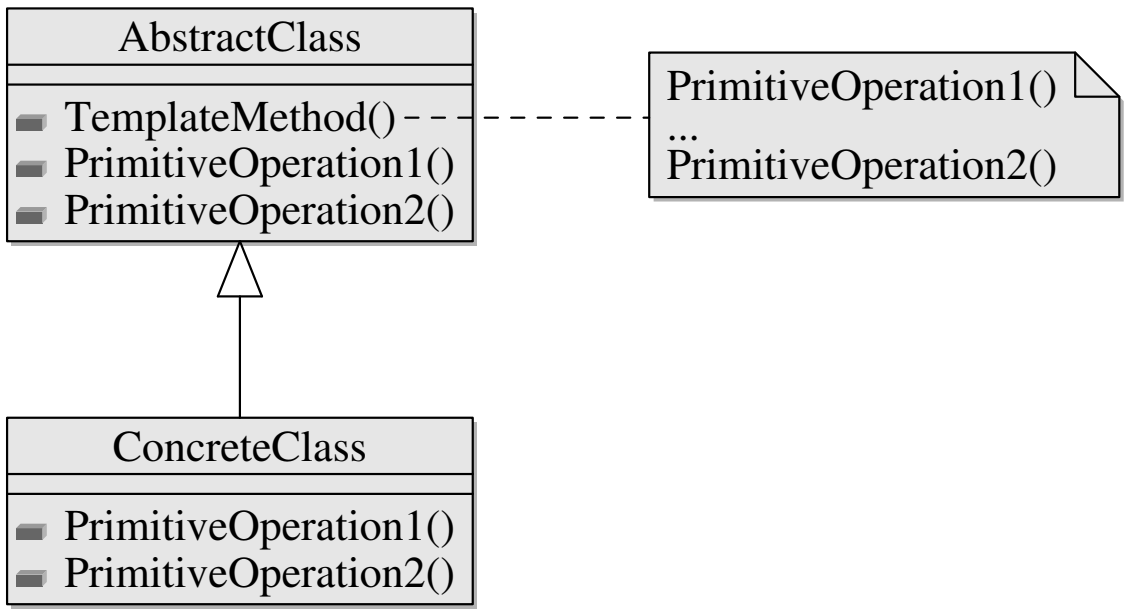


Image provenant de `http://thor.info.uaic.ro/~ogh/lop/lop.pdf`

Template Method : exemple `Arrays.sort()`

La méthode `Arrays.sort()` de l'API Java permet de trier des tableaux d'objets qui implémentent l'interface `Comparable` et ont donc une méthode `public int compareTo(Object object)`.

- `Arrays.sort()` est une méthode template.
- Elle utilise l'opération `public int compareTo(Object object)`.

Template Method avec hook

Hook

Un hook est une opération qui est déclarée dans la classe abstraite et qui peut être définie avec un comportement par défaut.

- Les classes dérivées peuvent ne pas l'utiliser.
- Les classes dérivées peuvent la définir.

Template Method avec hook : exemples

- La méthode `void paint(Graphics graphics)` de la classe `JFrame` est un hook.
- Les méthodes `void init()`, `void start()`, `void stop()`, `void destroy()` et `void paint(Graphics graphics)` de la classe `Applet` sont des hooks.

Strategy

Stratégie

Encaspsuler une famille d'algorithmes pour les rendre interchangeables. Permettre de modifier un algorithme indépendamment des clients.

- Différence avec le pattern Template Method : les algorithmes peuvent être complètement différents.
- On peut utiliser Template Method et Strategy ensemble.

Strategy

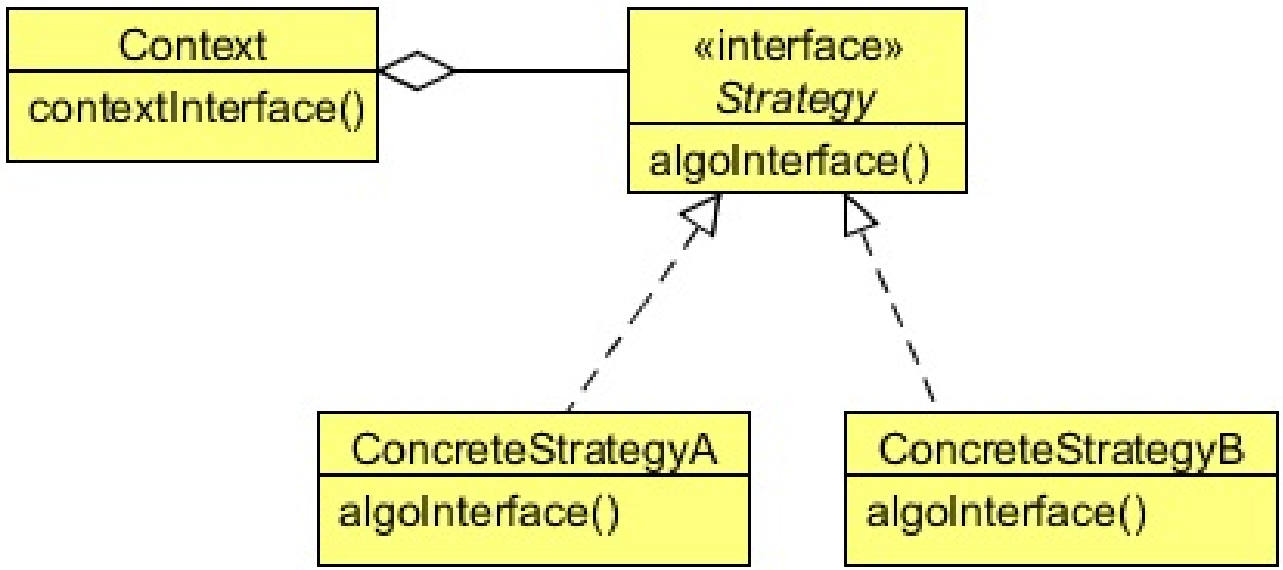


Image provenant de `http://www.javacodegeeks.com/2015/09/strategy-design-pattern.html`

Strategy : exemple

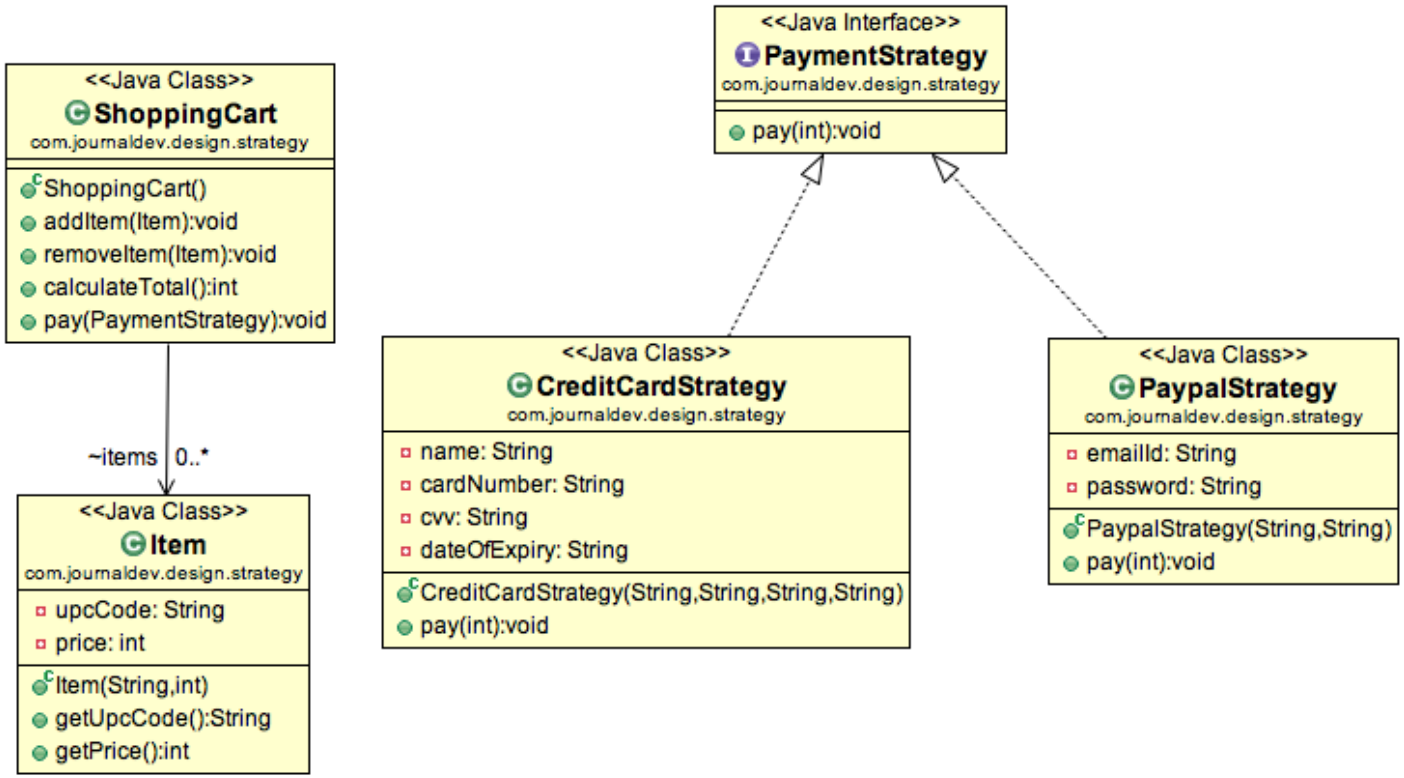


Image provenant de

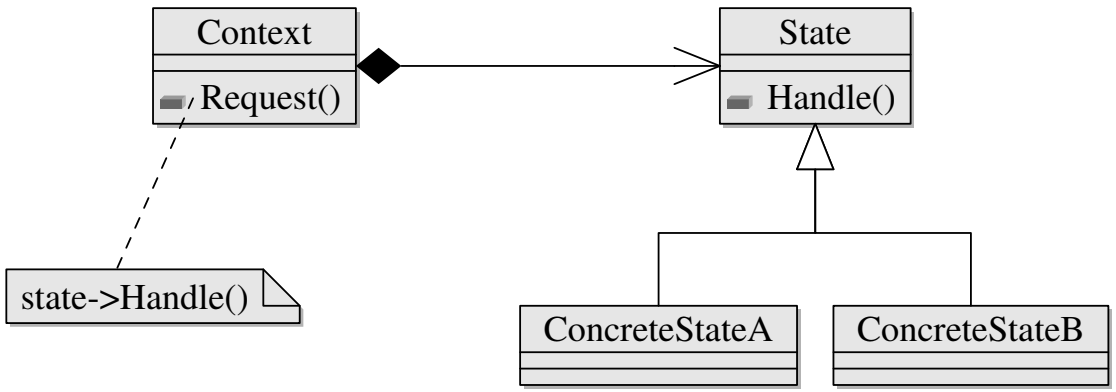
`http://www.javacodegeeks.com/2013/08/strategy-design-pattern-in-java-example-tutorial.html`

State

État

Modifier le comportement d'un objet lorsque son état change. L'objet semble changer de classe.

State

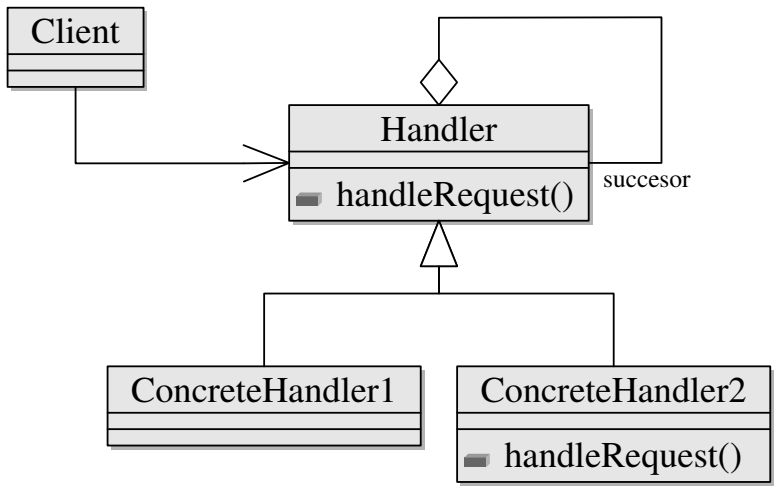

Context
Request()
State
Handle()
state->Handle()
ConcreteStateA
ConcreteStateB

Chain of Responsability

Chaîne de responsabilité

Éviter de coupler l'émetteur et le destinataire d'une requête en permettant à plusieurs objets de la prendre en charge. Les objets sont chaînés jusqu'à l'objet qui traite la requête.

Chain of Responsability

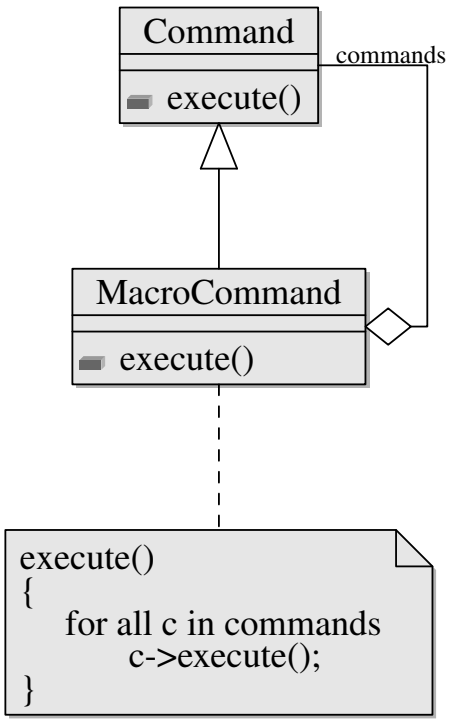


Command

Commande

Encapsuler une requête dans un objet. Permettre de paramétrer les clients avec plusieurs requêtes, files d'attente, historique, d'assurer le traitement des requêtes réversibles.

Command

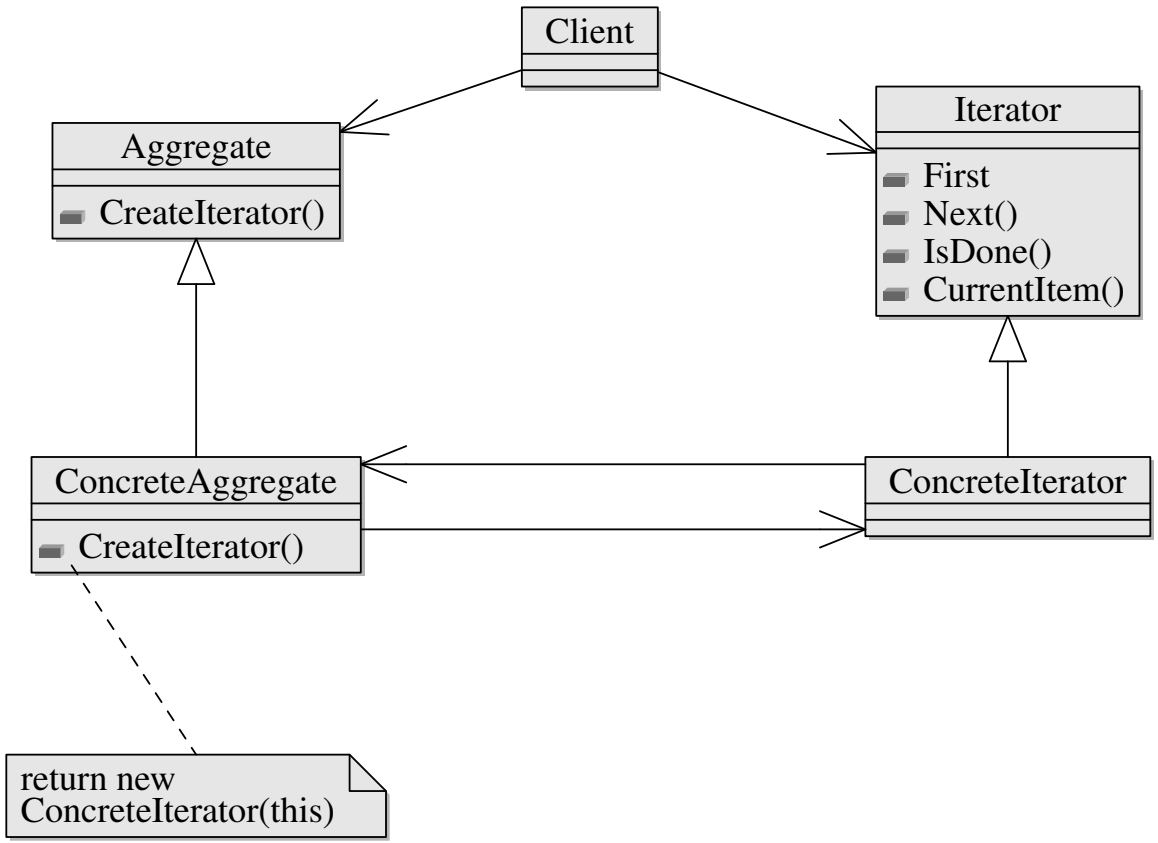

Command
commands
execute()
MacroCommand
execute()
execute()
{
for all c in commands
c->execute();
}

Iterator

Itérateur

Accéder en séquence aux éléments d'un objet de type agrégat sans révéler sa représentation sous jacente.

Iterator

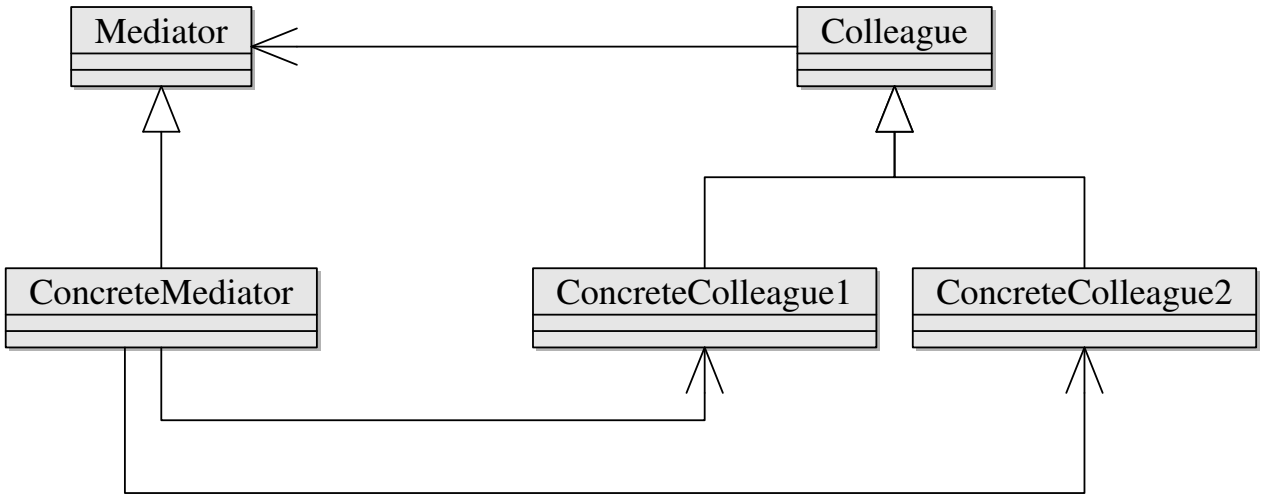

Client
Aggregate
CreateIterator()
Iterator
First
Next()
IsDone()
CurrentItem()
ConcreteAggregate
CreateIterator()
ConcreteIterator
return new
ConcreteIterator(this)

Mediator

Médiateur

Encapsuler les modalités d'intéraction entre objets pour favoriser les couplages faibles. Permettre de modifier une relation indépendamment des autres.

Mediator

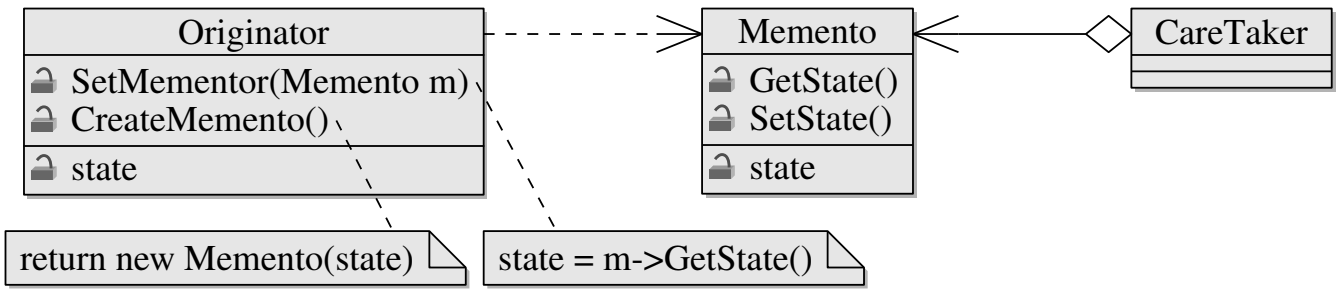

Mediator
Colleague
ConcreteMediator
ConcreteColleague1
ConcreteColleague2

Memento

Mémento

Acquérir et délivrer une information sur un état interne d'un objet sans violer l'encapsulation pour pouvoir rétablir l'objet dans cet état ultérieurement.

Memento


Originator
SetMementor(Memento m)
CreateMemento()
state
Memento
GetState()
SetState()
state
CareTaker
return new Memento(state)
state = m->GetState()

Model View Controller

Modèle Vue Controller

Séparer le modèle, noyau fonctionnel qui gère les données de l'application, de la vue, composants qui affichent les données pour l'utilisateur et du du contrôleur qui envoie les requêtes de l'utilisateur à la vue et au modèle.

- La liaison entre la vue et le modèle utilise le modèle Observer.
- La liaison entre la vue et le contrôleur utilise le modèle Strategy.
- La vue utilise le modèle Composite.
- Le contrôleur utilise le modèle GRASP Controller.

Model View Controller

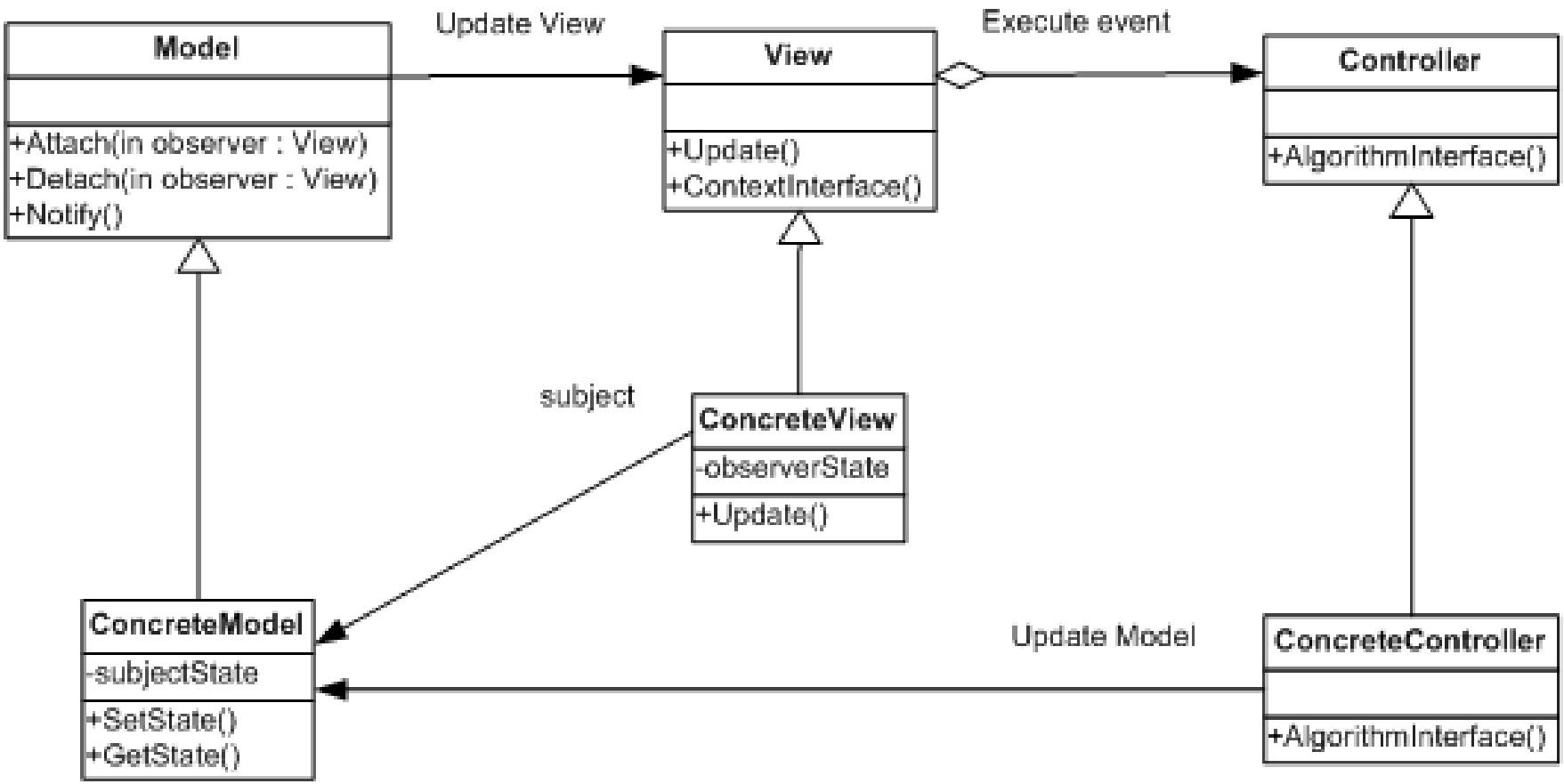


Image provenant de

Se préparer pour l'examen

- Connaître les bonnes pratiques de la programmation objet.
- Pour chaque pattern dont on a vu une implémentation en TP :
  - connaître l'objectif du pattern ;
  - savoir l'utiliser pour concevoir du code java ;
  - savoir l'utiliser pour réécrire du code existant ;
  - réfléchir à ses avantages ;
  - réfléchir à ses inconvénients et à ses limites.