SLR202
Patrons de conception
(Design Pattern)

Etienne.Borde@telecom-paristech.fr
Sylvie. Vignes@telecom-paristech.fr

Télécom ParisTech
Département Informatique et Réseaux

2014

Objectif et référence

- Construire ses logiciels en tenant compte de l'expérience collective des développeurs de logiciels
  - Réutiliser des solutions bien connues pour résoudre des problèmes bien connus
- GoF (Gang of Four)
  - E. Gamma, R. Helm, R. Johnson, J. Vlissides: « *Design Patterns – Elements of Reusable Object-Oriented Software* ». Addison-Wesley, 1995.
    - Réédition en avril 2007

INF222-DP -2

Objectifs pédagogiques

- Présenter quelques patrons de conception

I. Les *patterns* de construction
II. Les *patterns* de structure
III. Les *patterns* de comportement

Remarque : Il existe d'autres catégorisations

D'un problème à un pattern exemple

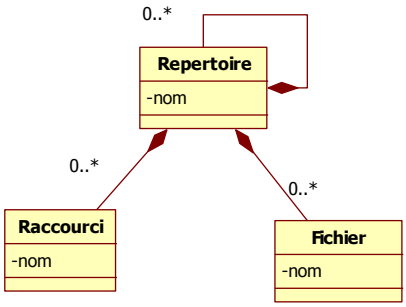
- Enoncé: proposer une solution élégante qui permette de modéliser le système de gestion de fichiers suivant:

1. Les fichiers, les raccourcis et les répertoires sont contenus dans des répertoires et possèdent un nom
2. Un raccourci peut concerner un fichier ou un répertoire
3. Au sein d'un répertoire donné, un nom ne peut identifier qu'un seul élément (fichier, sous-répertoire ou raccourci)

(ref livre : UML 2 par la pratique, P. Roques)

Exemple : phrase 1

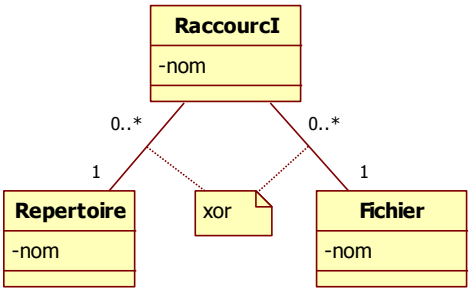
- 3 concepts => 3 classes
  - Contenance modélisée par une composition
  - Multiplicité côté contenant 1
  - Destruction répertoire entraîne destruction de tout ce qu' il contient



INF222-DP -5

Exemple : phrase 2

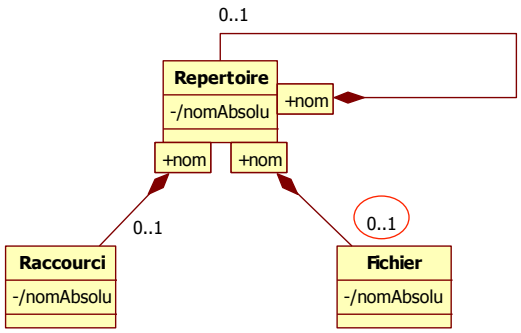
- un raccourci peut concerner un fichier ou un répertoire
  - 2 associations en exclusion mutuelle



INF222-DP -6

Exemple : phrase 3

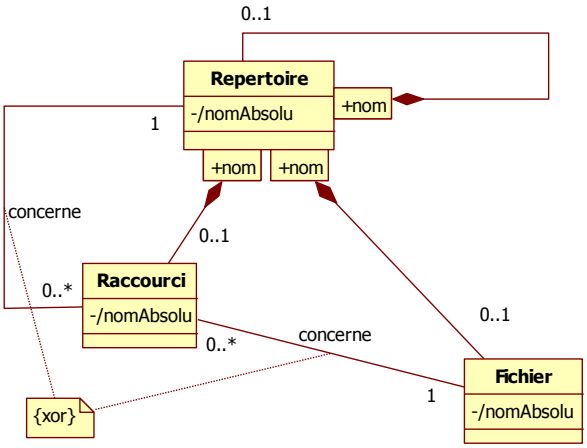
- Au sein d'un répertoire donné, un nom ne peut identifier qu'un seul élément
  - Idée : qualifier chacune des 3 compositions avec un attribut nom



INF222-DP -7

Premier modèle

- Pas tout à fait correct.
- Pourquoi?



INF222-DP -8

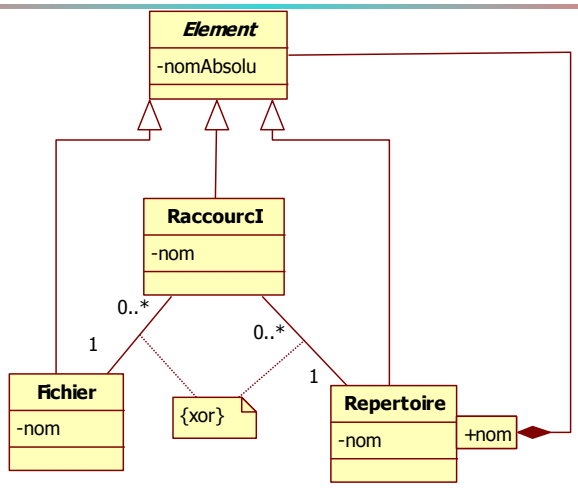
Exemple : discussion

- Pb : rien n'empêche qu'un fichier et qu'un raccourci aient le même nom
- 3 compositions qualifiées …
- Il faut un qualificatif unique pour chaque type d'élément contenu dans un répertoire

- Indices pour une solution :
  - Le terme élément est important
  - En faire une classe abstraite
  - Utiliser l'héritage
  - Modifier le modèle afin de n'avoir qu'une composition à qualifier

INF222-DP -9

Exemple : Solution

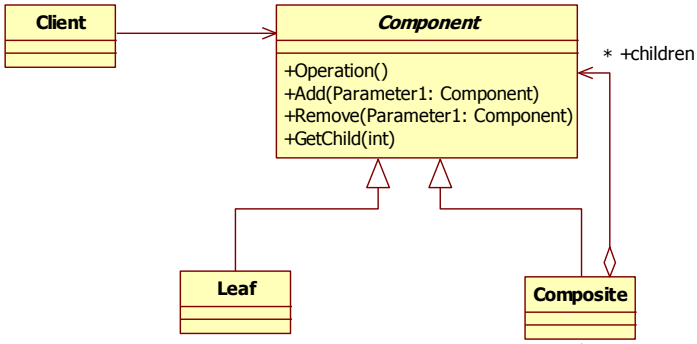


- Remarque
  - Introduction d'une classe abstraite
  - Double relation asymétrique entre *Repertoire* et *Element*
    - *Repertoire* contient des *Element*
    - *Repertoire* est un *Element*

INF222-DP -10

D'un problème à un pattern

- On a (re)trouvé le pattern Composite
  - Solution pour représenter des hiérarchies composant/ composé
  - Le « client » peut accéder et traiter de la même façon les objets individuels (feuilles) et leurs combinaisons (composites)

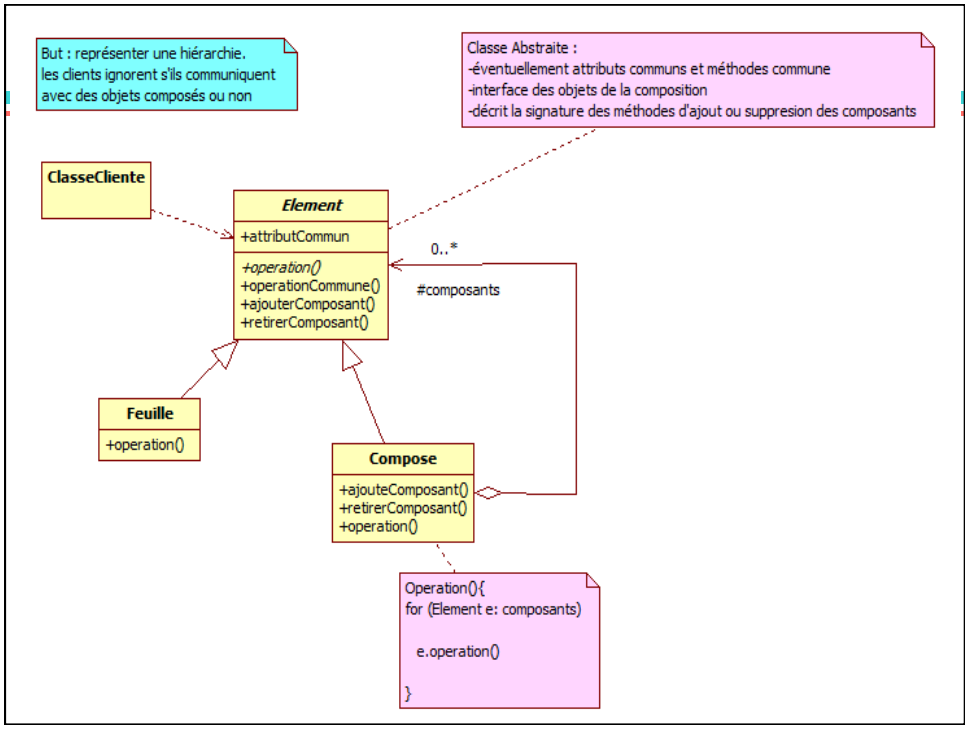


INF222-DP -11

Remarque : en Java
Différences entre *Classe Abstraite* et *Interface*

- Une classe peut implémenter autant d'interfaces que nécessaire mais elle ne peut étendre au plus qu'une classe abstraite.
- Une classe abstraite peut avoir des méthodes non abstraites, mais toutes les méthodes d'une interface sont abstraites.
- Une classe abstraite peut déclarer et utiliser des attributs alors qu'une interface ne le peut pas, bien qu'elle puisse créer des constantes static final.
- Une classe abstraite peut avoir des méthodes déclarées *public, protected, private ou sans accès (package)*. Les méthodes d'une interface ont un accès implicitement public.
- Une classe abstraite peut définir des constructeurs alors qu'une interface ne le peut pas.

INF222-DP -12



Un patron de conception

- Fournit une solution générique à une famille de problèmes
- Est décrit par
  - **Nom**
    - Significatif (quasi consensus dans les différentes catégorisations)
  - Problème
    - Décrit quand appliquer le modèle
    - Explique le problème et son contexte
  - Solution
    - configuration décrite en UML
      - Aspect **structurel**: diagramme de classes
      - Solution **abstraite** à appliquer
  - Avantages et inconvénients, usages connus (motivation, scénarios), patterns en relation …

TELECOM ParisTech

INF222-DP -14

Principales catégories

I. Les *patterns* de construction
- Abstraire le processus d' instanciation des objets
  Singleton, Factory Method, Abstract Factory …

II. Les *patterns* de structuration
- Organiser la hiérarchie des classes et organiser les relations entre classes

III. Les *patterns* de comportement
- Organiser les interactions entre objets et répartir les traitements entre objets …

I-Les patterns de construction

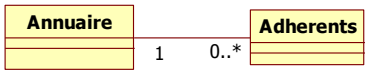
- Plus précisément
  - Abstraire le processus d'instanciation des objets.
  - Le rendre indépendant de la façon dont les objets sont créés, composés, assemblés, représentés.
  - Encapsuler la connaissance de la classe concrète qui instancie.
- Singleton, Factory Method, Abstract Factory …

I- *pattern* de construction

Singleton

- Garantir qu'une classe n'a qu'une seule instance et fournir une méthode de classe unique retournant cette instance

- Exemple :
  un annuaire référence les données des adhérents
  - Point d'entrée principal pour accéder à l'ensemble des instances
  - Faire en sorte que cet annuaire soit accessible à l'ensemble de l'application



  - Solutions
    - Naïve et contraignante : passer le répertoire en paramètre chaque fois
    - Avoir une seule instance de la classe accessible via une méthode statique

INF222-DP -17

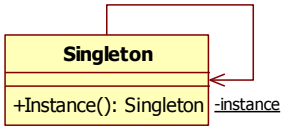
I- *pattern* de construction

Code Java Singleton : une solution

instanciation tardive sans possibilité d'héritage

```
public class Annuaire {
    private ArrayList annuaire = new ArrayList();
    public ArrayList getAnnuaire () { return annuaire } ;
    public ...

    private static Annuaire singleton = new Annuaire();
    public static Annuaire get() {
                return singleton;
        }
    private Annuaire() { }
}
```



Utilisation:

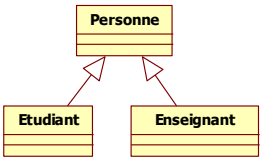
```
Annuaire annuaireUnique = Annuaire.get();
```

INF222-DP -18

I- *pattern* de construction

l'idée de la *Fabrique* (-> Factory method)

- Problème : la création d'un Etudiant et d'un Enseignant est différente (ex pas les mêmes attributs)



- Solution
  - Définir la méthode create abstraite au niveau de *Personne*
  - *Etudiant* et *Enseignant* fournissent leur propre create()
  - Revient à « virtualiser » le constructeur
  - Utiliser le polymorphisme

INF222-DP -19

I- *pattern* de construction

Code java

```
abstract class Personne {
    abstract Personne create();
...
}

class Etudiant {
    Personne create() {
    return new Etudiant();
} }

class Enseignant {
    Personne create() {
    return new Enseignant();
} }
```

classe cliente:

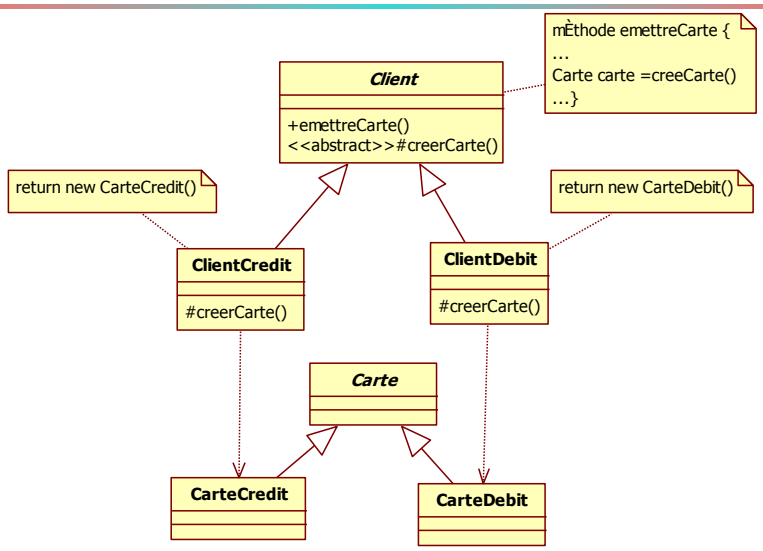
```
Personne p ;    // un étudiant ou un enseignant
Personne unePersonne = p.create();
                // unePersonne est du même type que p
```

INF222-DP -20

Exo Création de cartes de paiement

- Selon 2 catégories de clients
  - Ceux qui ont droit au crédit,
  - Ceux qui n'y ont pas droit
- Lors de leur demande d'une carte de paiement,
  - les premiers reçoivent une carte de crédit cad à débit différé sur le compte
  - les seconds peuvent seulement avoir une carte de débit cad à débit immédiat sur le compte.

La solution ou l'application d'un DP



I- *pattern* de construction

Pattern Factory method

- Contexte
  - création d'objets par des sous-classes
- Problème
  - une classe ne peut anticiper la classe d'objets à créer
  - Exemple :
    - une application sait quand elle doit créer un document (choix de l'utilisateur dans un menu)
    - mais elle ne sait pas quel type de document (l'utilisateur choisit dans le menu le type de document).
- Objectif («Intention») :
  - laisser un autre développeur définir l'interface permettant de créer un objet, tout en gardant un contrôle sur le choix de la classe à instancier

INF222-DP -23

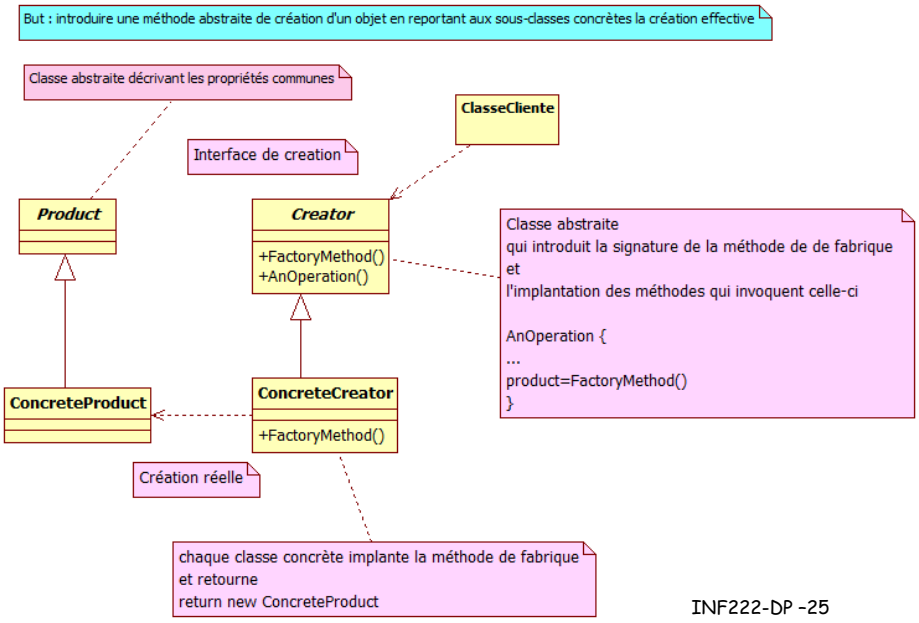
I- *pattern* de construction

Pattern Factory Method (description 2)

- Solution
  - Définir une **interface** pour créer un objet. Ce sont les sous-classes qui instancient réellement l'objet.
  - Structure et Collaborations
    - Une interface Product qui définit les objets à créer
    - Une classe ConcreteProduct qui implante l'interface Product
    - Une classe abstraite Creator, qui ne sait pas à l'avance de quelle classe sont les objets qu'il faut créer et qui définit une méthode abstraite, la Factory méthode, pour créer ces objets et des méthodes qui font appel à la Factory méthode.
    - Une classe ConcreteCreator qui implante la Factory méthode de sorte qu'elle retourne une instance de ConcreteProduct. Il peut y avoir plusieurs *ConcreteCreator*

INF222-DP -24

DP Factory Method



INF222-DP -25

Exo Création selon 2 modèles de cartes

- De plus, il existe 2 modèles de cartes de débit et de crédit, à savoir
  - les cartes Visa
  - les cartes MasterCard

INF222-DP -26

I- *pattern* de construction

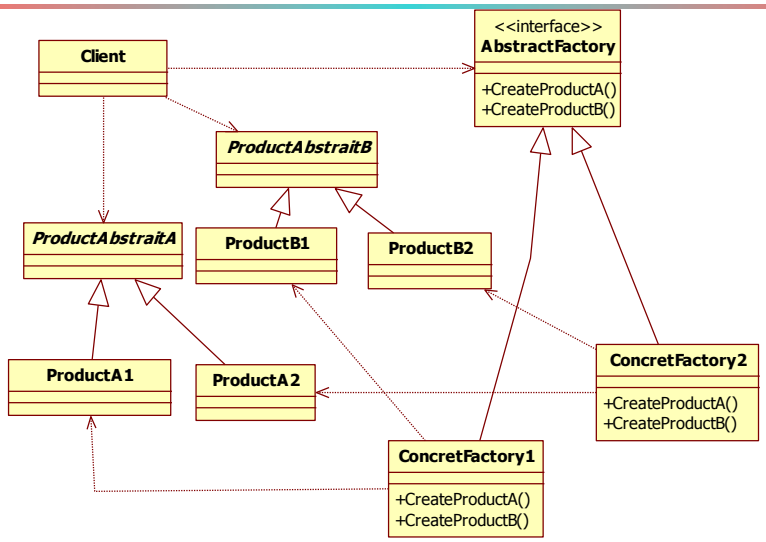
Abstract Factory

- Permettre la création de famille d'objets ayant un lien ou interdépendants sans connaître les classes concrètes de création
- Exemple:
  - Un site de vente de véhicules gère des automobiles et des scooters. Ces véhicules fonctionnent soit à l'essence soit à l'électricité. On veut un catalogue de tous les véhicules.
    - 2 hiérarchies de classes interdépendantes
- Généralisation de Factory Method
- La virtualisation de la création est appliquée à plusieurs hiérarchies

INF222-DP -27

I- *pattern* de construction

Le pattern Abstract Factory



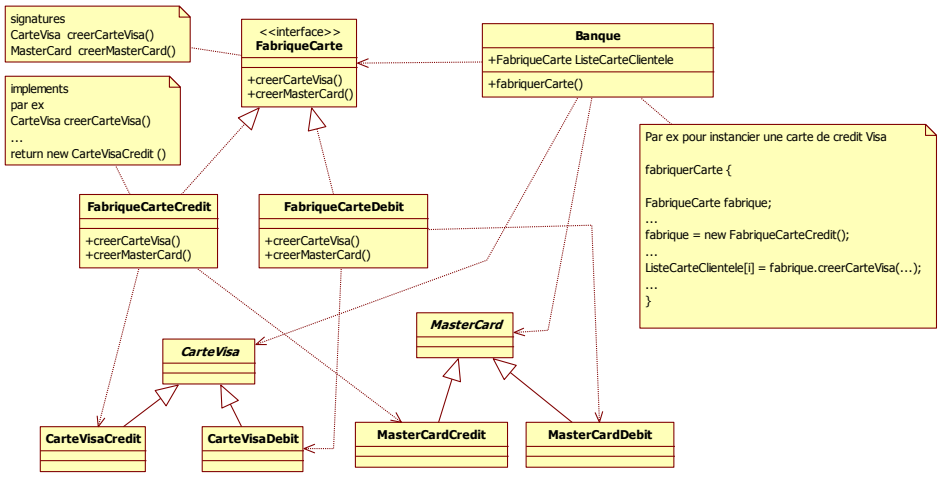
INF222-DP -28

Classes participant au DP

- Client : classe qui utilise l' interface de AbstractFactory
- AbstractFactory (FabriqueCarte) : interface spécifiant les signatures des méthodes créant les différents produits
- ConcretFactory1 [&2] (FabriqueCarteCredit) [&Debit] : classes concrètes implémentant les méthodes de création de produits pour chaque famille. Connaissant la famille et le produit, elles sont capables de créer une instance du produit de la famille
- ProductAbstractA [&B] (CarteVisa & carteMaster) sont des classes abstraites indépendamment de leur famille. Les familles sont introduites dans leur sous-classes concrètes

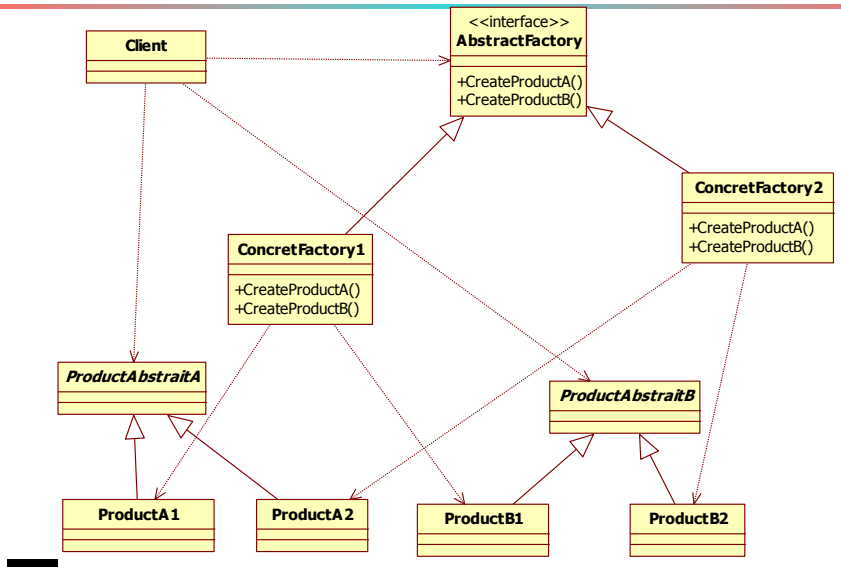
INF222-DP -29

Application du DP



INF222-DP -30

Le même DP !

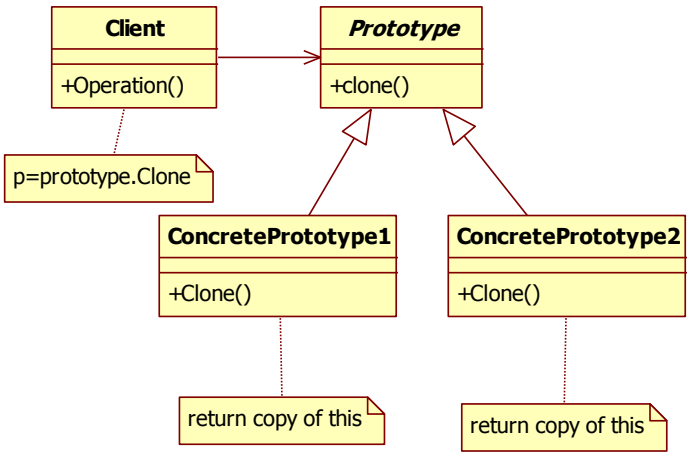


Pattern Prototype

- Création de nouveaux objets par duplication d' objets existants appelés prototypes qui disposent de la capacité de clonage
- C' est une instance qui sert de modèle
- On introduit une classe abstraite qui introduit une méthode duplique()
  - Un intérêt possible: créer de nouvelles instances initialisées comme le prototype

I- *pattern* de création

Le pattern Prototype



INF222-DP -33

Principales catégories

I. Les *patterns* de construction

II. Les *patterns* de structuration
- structure de classes pour séparer interface et implémentation, dans les « grands » systèmes

Composite, Adapter, Proxy, Bridge

III. Les *patterns* de comportement

INF222-DP -34

II- *pattern* de structure

ADAPTER

- Ailleurs classé *pattern d'interfaces*
- Contexte
  - Fournir l'interface qu'un client attend en utilisant les services d'une classe dont l'interface est différente
  - Permettre la collaboration d'instances dont les classes sont incompatibles
- Problème
  - Interfaces incompatibles
- Solution

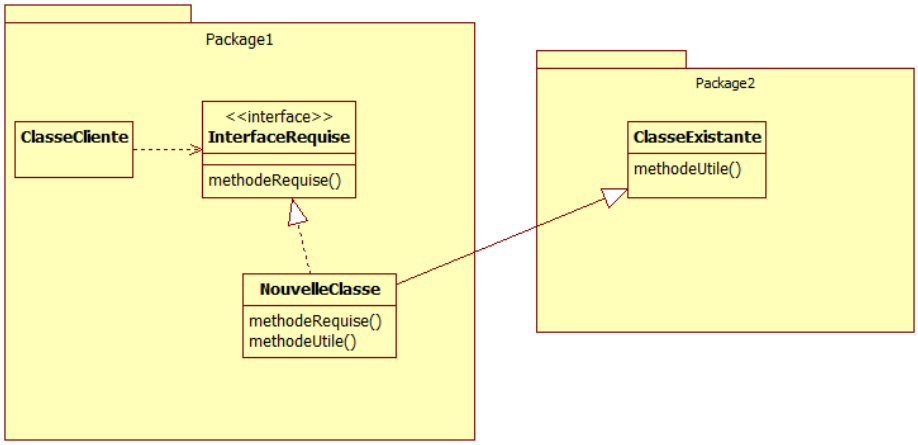
  2 variantes
  - Adaptateur de classe
  - Adaptateur d'objets

INF222-DP -35

II- *pattern* de structure

Adaptateur de classes

- Implémente l'interface désirée et étend une classe existante

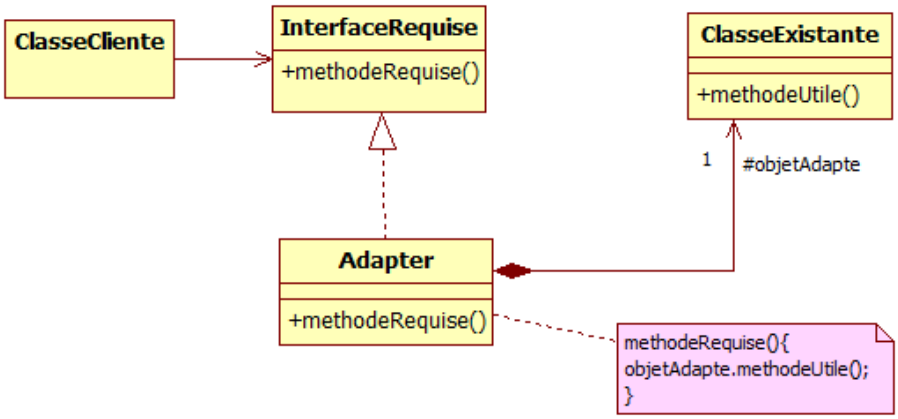


INF222-DP -36

II- *pattern* de structure

Adaptateur d'objet

- Utilise la délégation plutôt que la dérivation de sous-classes



INF222-DP -37

II- *pattern* de structure

Proxy

- Conception d'un objet qui se substitue à un autre objet (sujet) et qui en contrôle d'accès.
- Le proxy reçoit les appels du client à la place du sujet réel
- Concept très utilisé en informatique répartie
  - accès à des objets distants
  - caches
- proxy
  - "coquille" creuse
  - implémente la même interface que l'entité principale
  - possède une référence sur le sujet
  - code : appel des méthodes du sujet

INF222-DP -38

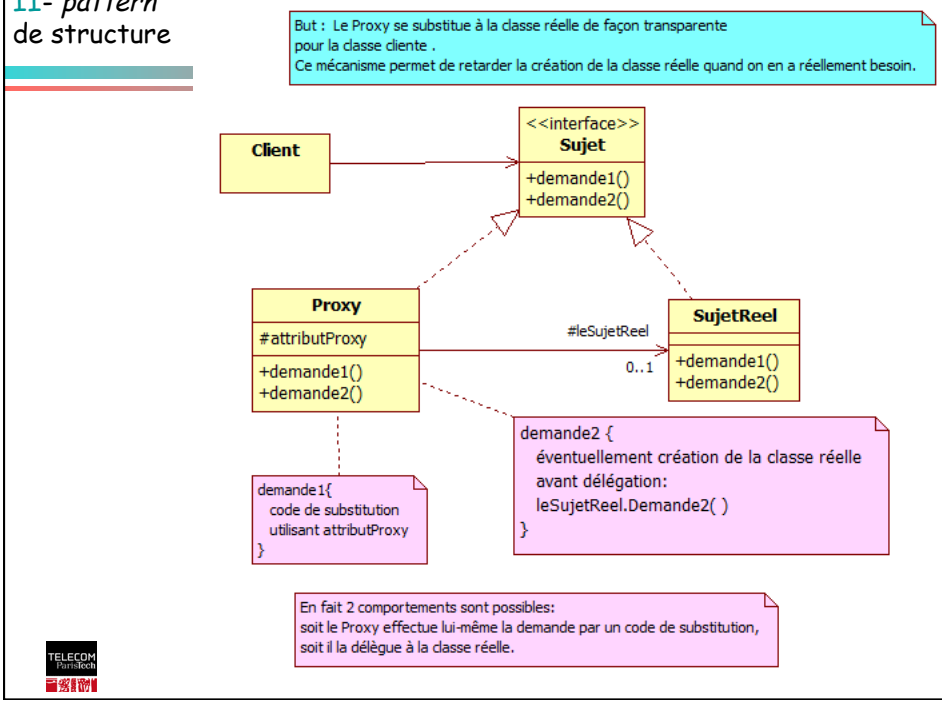
II- *pattern* de structure

Description de Proxy

- Contexte
  - Accéder indirectement à des objets
- Problème
  - On a besoin d ' accéder à un objet mais on ne veut pas d ' un pointeur direct sur l' objet (raisons d'efficacité ou bien de sécurité)
  - Exemple:
    - La création d ' un objet qui "coûte cher" est repoussée au moment où l ' on a vraiment besoin de l ' objet.
    - Ouvrir un document sans ouvrir immédiatement tous les graphiques qu' il contient. Attendre d ' arriver sur les pages qui contiennent des graphiques.

INF222-DP -39

II- *pattern* de structure



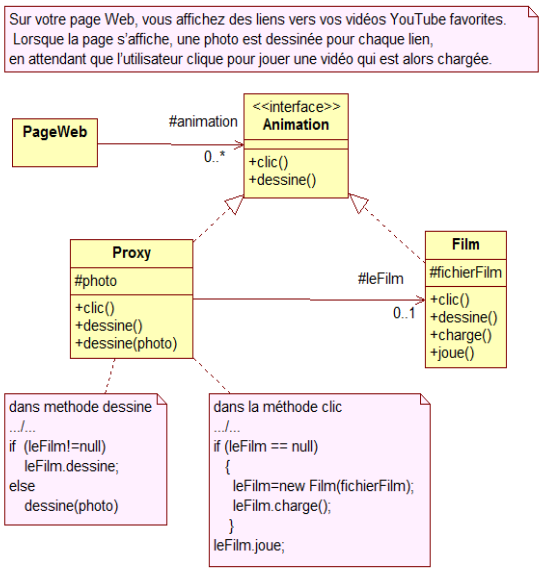
II- *pattern* de structure

Description de Proxy

- Solution
  - Contrôler l ' accès à un objet en utilisant un objet intermédiaire: le proxy
  - Structure et Collaboration
    - Une interface Subject qui est une interface commune au sujet réel et au proxy
    - Une classe Proxy qui maintient une référence sur l' objet réel, et contrôle l' accès à l' objet réel
    - Une classe RealSubject qui implante l ' objet réel représenté par le proxy
    - Le client accède au proxy qui lui-même accède ensuite à l ' objet réel

INF222-DP -41

Exemple Pattern Proxy

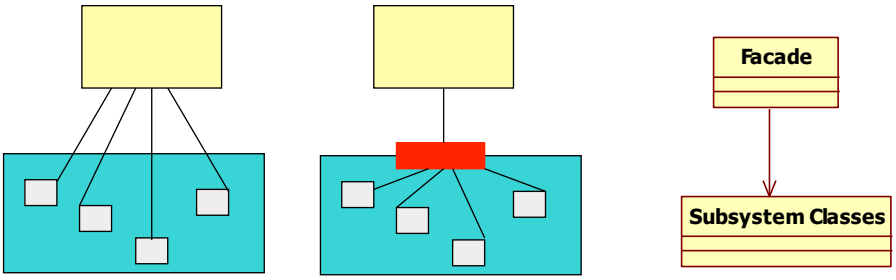


!2-DP -42

II- *pattern* de structure

FACADE

- fournit un point d'entrée commun pour l'accès aux éléments d'un sous-système
  - simplifie l'accès aux librairies complexes
  - masque les détails de réalisation du sous-système
  - inconvénient : la façade doit évoluer avec le sous-système
  - Remarque : sert beaucoup pour composant logiciel JavaEE

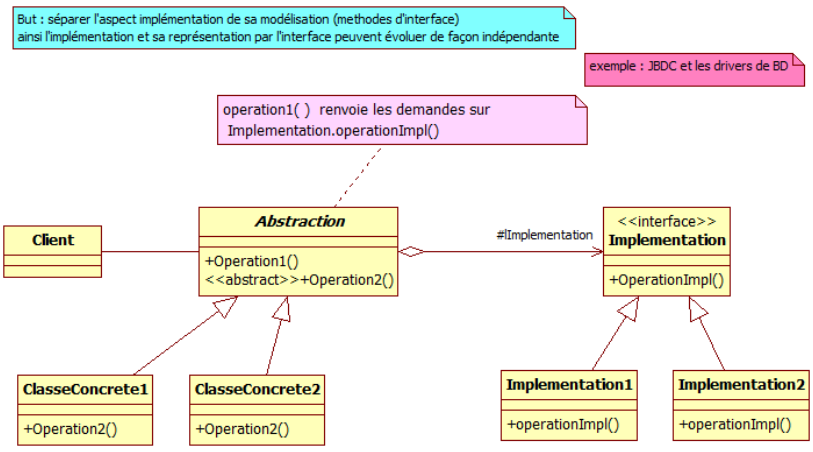


INF222-DP -43

II- *pattern* de structure

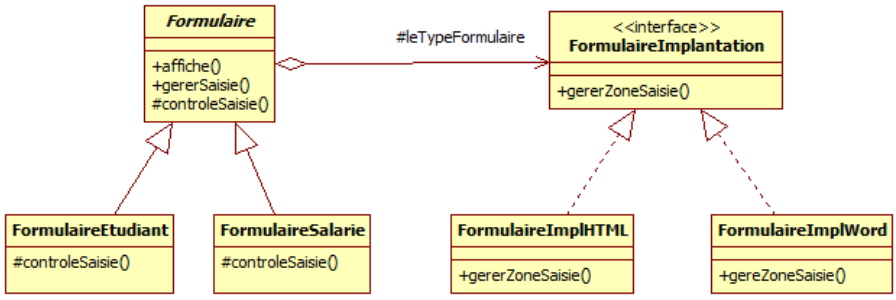
Pattern Bridge

- Découpler une classe qui s'appuie su des opérations abstraites de l'implémentation de ces opérations pour les faire varier séparément



INF222-DP -44

DP Bridge Exemple

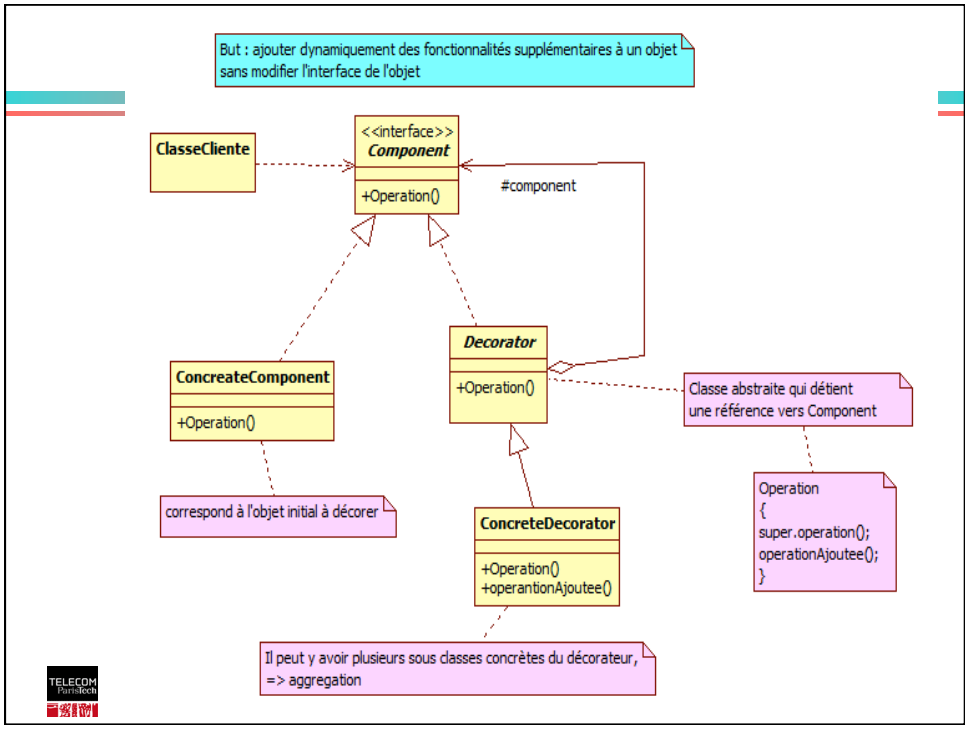


INF222-DP -45

Pattern Decorateur

- But : ajouter dynamiquement des fonctionnalités supplémentaires à un objet
- C' est une alternative à la création d' une sous-classe pour enrichir un objet
- exemple classique:
  - en Java flux d'E/S et objets Writer

INF222-DP -46



III- *pattern* de comportement

Principales catégories

I. Les *patterns* de construction

II. Les *patterns* de structure

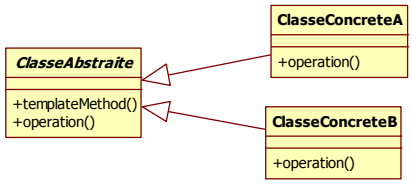
III. Les *patterns* de comportement
- Interaction entre classes
  - = gérer des algorithmes
  - = répartir les « responsabilités »

Iterator, Observer, MVC

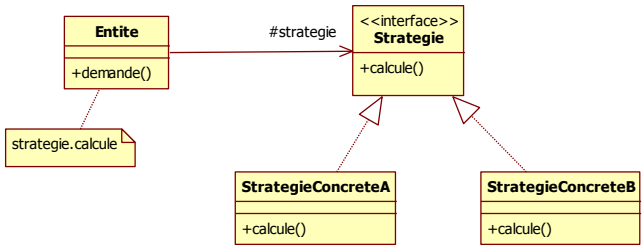
INF222-DP -48

Dans ces patterns, 2 approches

- Héritage



- Délégation



TELECOM ParisTech

INF222-DP -49

III- *pattern* de comportement

Template Method (Patron de méthode)

- Contexte
  - Partager des algorithmes uniques dont l'implémentation concrète peut varier
- Problème
  - Comment implanter les parties invariantes d'un algorithme et laisser les sous-classes implanter les parties variables
  - Exemple:
    - l'ouverture d'un document se fait en plusieurs étapes: le fichier peut-il être ouvert, créer un objet représentant le document (l'information dans le fichier) spécifique à l'application, lire le document.
    - Mais chaque étape est particulière aux documents et à l'application.

TELECOM ParisTech

INF222-DP -50

III- *pattern* de comportement

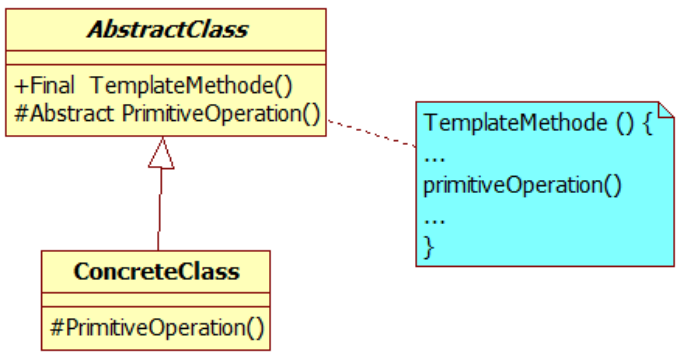
Template Method

- Solution
  - Définir le squelette de l' algorithme dans une opération et implanter les étapes de l ' algorithme dans les sous-classes
  - Structure et collaborations:
    - Une classe abstraite et plusieurs sous-classes concrètes
    - La classe abstraite définit une (ou plusieurs) Template méthodes (qui ne sont pas redéfinies par les sous-classes).
    - La classe abstraite définit une ou plusieurs méthodes abstraites Primitives (redéfinies par chacune des sous-classes). Une Template méthode fait appel aux méthodes Primitives.

INF222-DP -51

III- *pattern* de comportement

Template Method

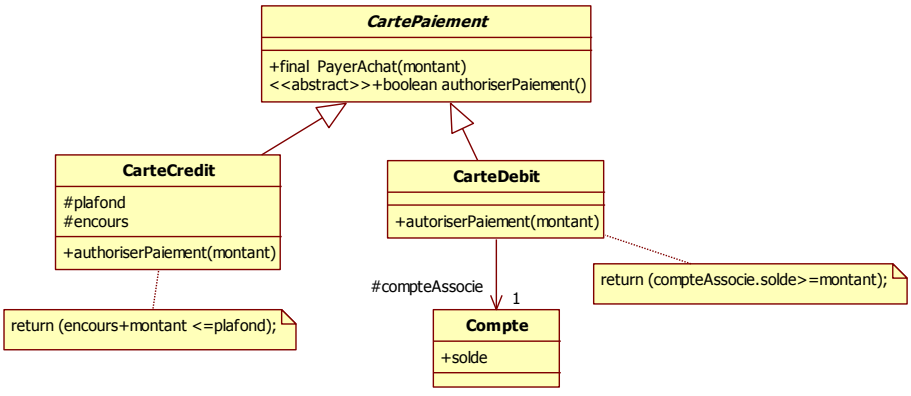


INF222-DP -52

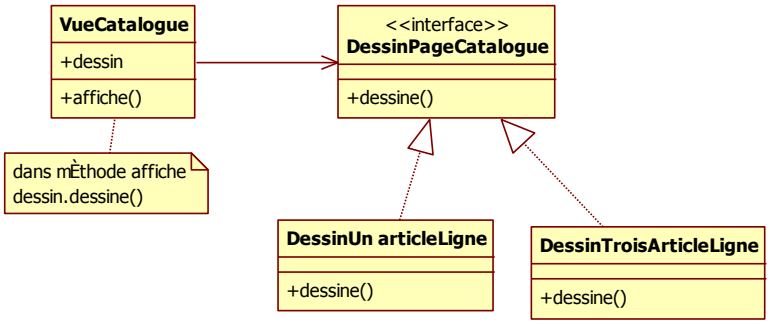
Classes participantes

- AbstractClass introduit une méthode « patron » ainsi que la signature des méthodes abstraites que cette méthode invoque
- La sous-classe concrète implémente les méthodes abstraites
- Il peut y avoir plusieurs classes concrètes

Exo Autorisation cartes de paiement

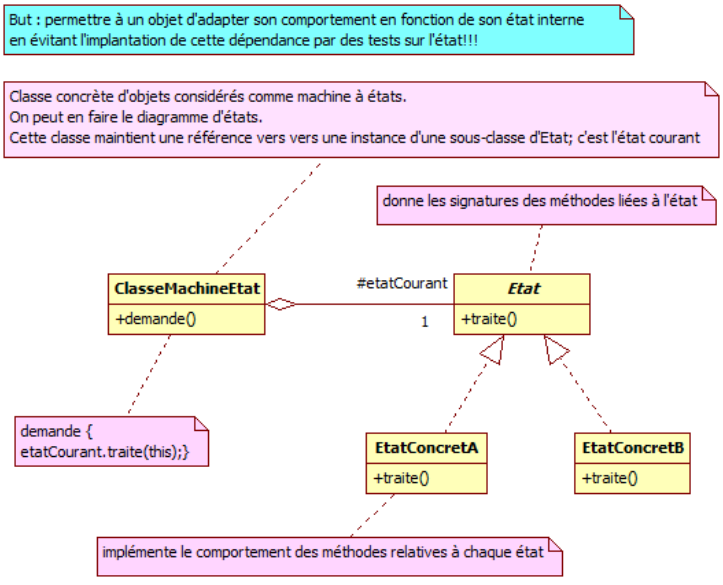


Pattern State par un exemple



INF222-DP -55

DP State



III- *pattern* de comportement

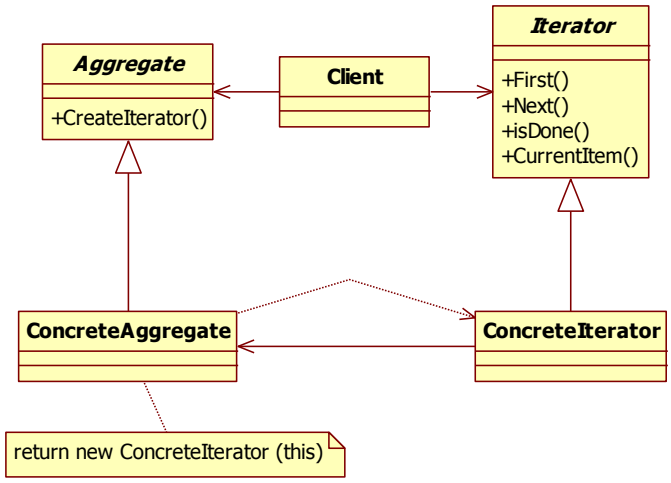
Iterator

- Permettre de parcourir séquentiellement une structure de données sans avoir à connaître cette structure
  - Java manipule naturellement les itérateurs
  - Java permet de définir des itérateurs pour des collections de données
    - 2 interfaces : java.util.Iterator et java.lang.Iterable
- Pattern????

INF222-DP -57

III- *pattern* de comportement

Le pattern Iterator

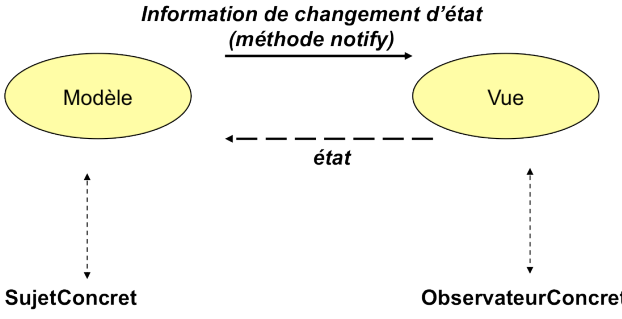


INF222-DP -58

III- *pattern* de comportement

Pattern Observer

- Répercuter les changements de l'état d'un objet vers un ou plusieurs autres objets
  - ex. : IHM, surveillance de données
- Le client est informé lorsqu'un objet change; il doit alors s'enquérir du changement.



INF222-DP -59

III- *pattern* de comportement

Pattern Observer (détails)

- un observé
  - produit des événements
  - gère une liste d'observateurs abonnés
  - notifie les observateurs lorsque un événement se produit
- un événement
  - peut être typé (≠ types d'événements)
  - a une source (observé)
  - peut être associé à des données (String message, double valeur, …)
    - ou peut être vide (observateur récupère les données sur la source)
- un observateur
  - peut être abonné à 1 ou plusieurs types d'événements
  - auprès d'1 ou de plusieurs observés

INF222-DP -60

III- *pattern* de comportement

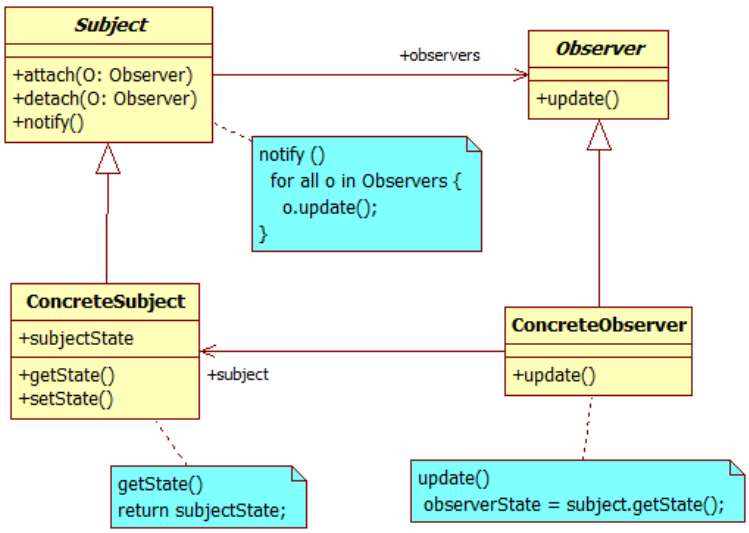
Description de Observer (Modèle-Vue)

- Contexte
  - Un composant utilise des données et informations procurées par un autre composant
- Problème
  - Définir des dépendances entre objets, de telle sorte que lorsqu' un objet change d' état, tous ceux qui en dépendent sont mis à jour automatiquement.
  - Il faut éviter que le composant fournissant les informations ne dépendent des autres composants.
  - Exemple: mêmes données avec plusieurs affichages différents

INF222-DP -61

III- *pattern* de comportement

Pattern Observer



III- *pattern* de comportement

Description de Observer (Modèle-Vue)

- Solution
  - Introduire un mécanisme de propagation des changements entre le détenteur d ' informations (Subject) et les composants qui en dépendent (Observers)
  - Structure et collaborations:
    - Une interface Subject, observée par les Observers et qui permet d ' attacher ou de détacher des Observers
    - Une interface Observer qui déclare une méthode update() de mise à jour des Observers
    - Une classe ConcreteSubject qui implante l ' objet observé, et qui informe les Observers lors de modifications
    - Une classe ConcreteObserver qui maintient une référence sur un objet de la classe ConcreteSubject et implante l ' Observer pour que son état soit cohérent avec celui du sujet

III- *pattern* de comportement

Exemple Pattern Observer

III- *pattern* de comportement

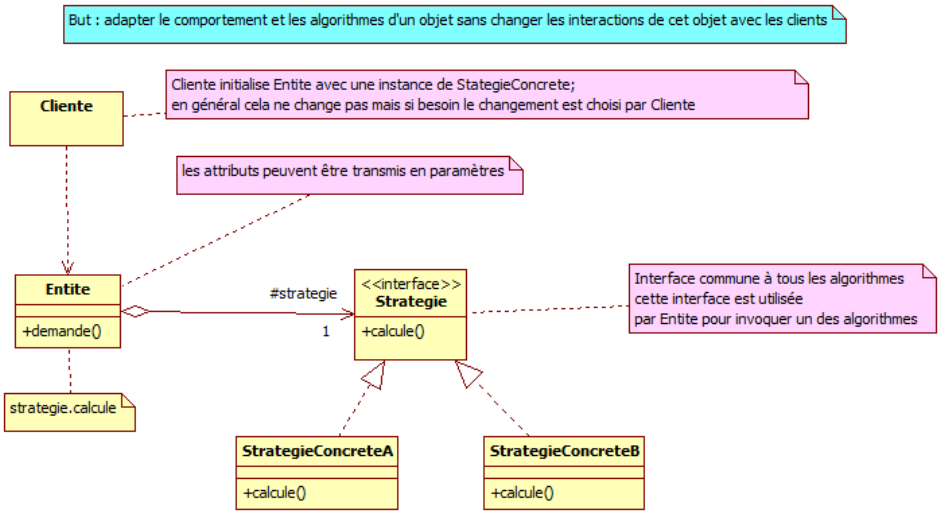
En référence au pattern MVC

- utilisé pour les IHM, plus généralement utilisé par frameworks Web, SI
- Modèle
  - la structure en mémoire stockant les données à afficher
- Vue
  - la représentation graphique de ces données (tableau, diagramme en barre, …)
- Contrôleur
  - assure le lien entre la (les) vue et le modèle
    - répercute les changements dans les données au niveau de la vue
    - en fonction des entrées de l'utilisateur au niveau de la vue, répercute les changements sur les données

Pattern Strategy

- Adapter le comportement et les algorithmes sans changer les interactions de cet objet avec les clients
- Il s'agit « d'aspects » (présentation, efficacité, représentation interne …) pas de besoins fonctionnels
- L'objectif du Pattern Strategy est d'encapsuler des approches ou stratégies alternatives dans des classes distinctes qui implémentent chacune une opération commune.

Pattern Strategy



INF222-DP -67

III- *pattern* de comportement

Pattern MVC

- Problème
  - supporter les différentes catégories d' interfaces graphiques
    - ex: différents affichages graphiques pour les mêmes statistiques
- Solution
  - Séparer les responsabilités relatives aux données, de celles relatives aux fonctionnalités, et à l' affichage
- Pattern
  - Modèle-Vue-Contrôleur : découplage des vues et du modèle en instaurant un protocole souscrit/notifie
- En fait combinaison de Patterns : Observer, Strategy …

INF222-DP -68

Conclusion

- Patterns => style architectural => organisation de code => efficacité
- La génération automatique de certains patterns est possible
  - Ex Proxy
- Certains patterns sont proches de construction de langage
- On explique le code des *framework* en désignant les patterns

INF222-DP -69