SLAM

Design Patterns & paradigmes

Design Pattern

* Quelques prérequis :

* POO : héritage, polymorphisme, classe abstraite, interface

* UML : diagramme de classe, héritage, implémentation, dépendance, classe abstraite, interface

Paradigmes de programmation

* D'après Stroustrup (C++), on avance !!!!
* Paradigme 0 (débutant) : tout dans le main, on test jusqu'à ce que ça marche !
* Paradigme 1 (programmation procédurale) : Découpage du main en sous-programme, paramètres, structure de contrôle correctement utilisée
* Paradigme 2 (programmation modulaire) : utilisation d'entêtes et d'implémentations, il est possible de masquer un partie des traitements

Paradigmes de programmation

* Paradigme 3 (abstraction de données) : utilisation des classes
* Paradigme 4 (héritage) : introduction du principe de substitution (polymorphisme, redéfinition, interface, implémentation, classe abstraite)
* Paradigme 5 (généricité) : templates et variadiques
* Paradigme final (productivité) : factoriser le code, mettre en oeuvre les différents paradigmes.

Paradigmes de programmation

* Paradigme final (productivité) : sous-entend l'utilisation de :
  * Interfaces : type abstrait, mise à disposition de classe
  * Design patterns : solutions classiques à un problème de programmation
  * Patterns d'architecture : MVC, …
  * Bibliothèques : de classes, de fonctions, …
  * Framework : architecture disponible dans un environnement, qui propose des patterns à compléter.

Principes de POO

* Inclut les principes de base de la programmation structurée :

1. Eviter la duplication du code : il faut factoriser ce qui se répète !
2. Recherchez la cohésion (forte) : il faut donc déjà réfléchir en amont ! Les composants d'un module, package, doivent avoir une unité logique forte.
3. Il faut séparer, la saisie, le calcul et l'affichage
4. Limiter la taille et la complexité : une méthode, une fonction devient longue dès qu'elle fait plus d'une tâche logique
5. Rechercher un couplage faible et forte cohésion afin de favoriser l'évolution du code.

Principes de POO

* Principes de base de la POO :

1. Encapsulation : non manipulation directe
2. Substitution : à travers l'héritage

Principes de la POO

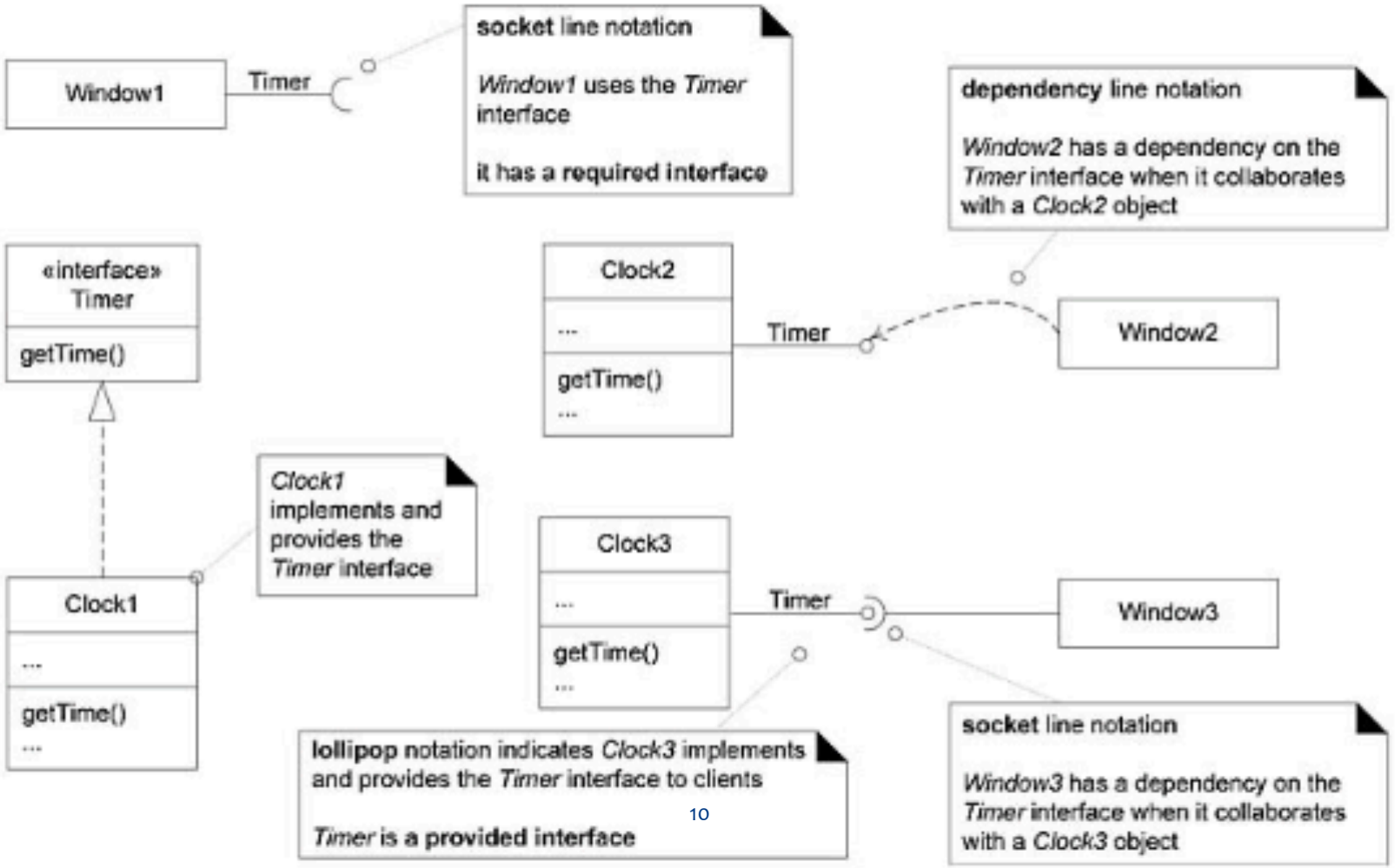
* Principes avancés de POO :
* **Qu'est-ce qu'une interface ?**
  * Classe sans attribut dont toutes les opérations sont abstraites
  * Ne peut être instanciée
  * Doit être réalisée (implémentée) par des classes non abstraites
  * Peut hériter d'une autre interface

Principes de la POO

* **Pourquoi des interfaces ?**
  * Utilisation similaire aux classes abstraites
  * En Java, en objC : une classe ne peut hériter de plus d'une classe, mais elle peut réaliser plusieurs interfaces
  * En C++, la limitation n'existe pas, les interfaces ne sont donc pas présentes

Principes de la POO

Exemple



Principes de POO

* Inclut les principes avancés de POO :

1. Encapsuler les attributs et les méthodes : Encapsuler ce qui varie permet de limiter les modifications du code à plusieurs endroits (méthodes dans paquetages)
2. Préférer la composition à l'héritage !
3. Coupler faiblement les classes, grâce aux interfaces. La factorisation des références se fait grâce aux interfaces.
4. Attribuer les bonnes responsabilités aux bonnes classes : difficulté de la POO et donc de l'analyse.

Notion de patterns

* Même genre de tâche, même genre de développement logiciel, même problème à résoudre = modèle existant, évolutif, flexible, REUTILISABLE
* Un framework est un pattern à moitié fini !

* Un pattern se veut normalement indépendant du langage et des environnements d'implémentation

Notion de patterns

* Les patterns ont un vocabulaire commun, ce qui leur permet l'implémentation suivant un langage.
* Chaque pattern possède un nom, ce qui permet de le repérer
* Un effort d'apprentissage est requis pour pouvoir les mettre en oeuvre.

Familles de patterns

* Design patterns : modèle de conception
* Patterns d'implémentation : propre à un langage
* Patterns d'architecture : MVC
* Patterns d'analyse : pattern métier correspondant à une domaine d'application (pattern générique des entêtes et des lignes en facturation)
* Patterns organisationnels : relatifs aux problèmes d'organisation des activités du développement logiciel

Design Patterns

* Patterns de conception les plus utilisés
* DCL (micro-architecture) fournissant des solutions à des problèmes répertoriés
* Ils utilisent le plus souvent héritage, interface, classe abstraite
* Il existe 23 design patterns :
  * 5 patterns de construction (création des objets)
  * 7 patterns de structuration (hiérarchie des classes et relations)
  * 11 patterns de comportement (répartition des traitements entre les objets)

	Stratégie	Comportement
	Etat	Comportement
	Observer	Comportement
	Decorateur	Structuration
	Composite	Structuration
	Itérator	Comportement
	Méthode de fabrique	Construction
	Fabrique abstraite	Construction
	Singleton	Construction
	Adapteur	Structuration
	Façade	Structuration
	Patron de Méthode	Comportement
	Proxy	Structuration
	Commande	Comportement

➢ *Les 14 + 1 patterns les plus populaires*	➢ *Les 9 design patterns restants*
• Stratégie • Observer • Decorateur • Méthode de fabrique • Fabrique abstraite • Singleton • Commande • Etat • Adapteur • Façade • Patron de méthode • Composite • Itérateur • Proxy +1 : MVC	• Chaine de responsabilités • Interprète • Médiateur Mémento Monteur Poids-mouche Pont Prototype Visiteur

Patterns de création

- Prototype
- Builder

Patterns structuraux

- Bridge
- Flyweight
- Proxy

Patterns comportementaux

- Visitor
- Chain of responsiblity
- Interpreter
- Mediator
- Memento
- Template method

N°	Type	Nom	Description (ou objectif ou définition) dite de plusieurs manières
1 ***	Comportement	Stratégie	• Encapsule des comportements d'une même famille (des méthodes plutôt que des attributs) et utilise la délégation pour savoir lequel utiliser. • « Sette » des méthodes à travers une interface. • Adapter le comportement d'un objet en fonction d'un besoin sans changer les interactions, et donc indépendamment du client. • Définir une famille d'algorithmes interchangeables et permettre de les changer indépendamment de la partie cliente. • Stratégie et Etat sont des patterns jumeaux.
2 ***	Comportement	Observer	• Définit une relation entre objets de type un-à-plusieurs, de façon que lorsque un objet change d'état, tous ceux qui en dépendent en soient notifiés et soient mis à jour automatiquement. • Met vos objets au courant quand il se passe quelque chose qui pourrait les concerner • Permet de notifier des changements d'état à des objets. • Construit une dépendance entre un sujet et des observateurs de sorte que chaque modification du sujet[18]soit notifiée aux observateurs afin qu'ils puissent mettre à jour leur état.

Les étoiles représentent le niveau de difficulté !

Les étoiles représentent le niveau de difficulté !

N°	Type	Nom	Description (ou objectif ou définition) dite de plusieurs manières
3 **	Structuration	Decorateur	• Donne de nouvelles responsabilités aux objets sans modifier les classes sous-jacentes. • Attache dynamiquement des responsabilités supplémentaires à un objet. Il fournit une alternative souple à la dérivation, pour étendre les fonctionnalités.
4 *	Structuration	Adapteur	• Adapteur et Façade sont des patterns jumeaux.
5 *	Structuration	Façade	• Adapteur et Façade sont des patterns jumeaux.
6 *	Comportement	Patron de Méthode	• Template Method permet de définir une méthode en déléguant une partie de l'algorithme dans des classes dérivées. La méthode est donc un « patron » concrétisé grâce aux classes dérivées.

N°	Type	Nom	Description (ou objectif ou définition) dite de plusieurs manières
7 ***	Construction	Méthode de fabrique	• Méthode de fabrique et fabrique abstraite sont des patterns complémentaires.
			• Le pattern Factory Method (pattern de fabrication ou méthode de fabrique ou patron de méthode de fabrique) a pour objectif de définir une classe de fabrique d'un produit avec une méthode de création abstraite du produit, la méthode abstraite étant concrétisée dans des classes de fabrication spécifiques. • Les patterns Template Method et Factory Method sont des patterns jumeaux.

Les étoiles représentent le niveau de difficulté !

N°	Type	Nom	Description (ou objectif ou définition) dite de plusieurs manières
8 ***	Construction	Fabrique abstraite	• Méthode de fabrique et fabrique abstraite sont des patterns complémentaires. • Fournit une interface proposant des méthodes de création de produits regroupés en familles. • La différence avec le pattern « Méthode de fabrique » est que la fabrique abstraite gère plusieurs familles de produits et qu'on gère une interface plutôt qu'une classe abstraite
9 *	Construction	Singleton	• Le pattern singleton permet de s'assurer qu'une classe ne possède qu'une seule instance.
10 ***	Comportement	Commande	• **E**ncapsule une requête comme un attribut permettant ainsi de faire fonctionner n'importe quel objet tout en en étant découplé. • Le pattern Commande et une fusion du pattern Stratégie (« Sette » des méthodes à travers une interface) et du pattern Adapteur (ajouter une nouvelle classe ou un nouveau composant à un modèle en faisant en sorte que l'interface du modèle ne change pas et donc que le client puisse continuer à fonctionner identiquement).

Les étoiles représentent le niveau de difficulté !

N°	Type	Nom	Description (ou objectif ou définition) dite de plusieurs manières
11 **	Comportement	Etat	• Stratégie et Etat sont des patterns jumeaux.
12 **	Structuration	Composite	• Itérateur et Composite sont des patterns jumeaux.
13 **	Comportement	Itérateur	• Itérateur et Composite sont des patterns jumeaux.
14 **	Structuration	Proxy	
+1 ***	Pattern de patterns	MVC	Obervateur (modèle), Stratégie (vue et contrôleur), Composite (vue).

Les étoiles représentent le niveau de difficulté !

Design Patterns

* patterns de construction (création des objets) :
  * *Abstract factory, Factory method, Singleton*
  * *Prototype, Builder*
* patterns de structuration (hiérarchie des classes et relations)
  * *Decorator, Adapter, Facade, Composite*
  * *Bridge, Flyweight, Proxy*
* patterns de comportement (répartition des traitements entre les objets)
  * *iterator, Strategy, State, Observer, Command*
  * *Visitor, Chain of responsiblity, Interpreter, Mediator,*
  * *Memento, Template method*

Design Patterns

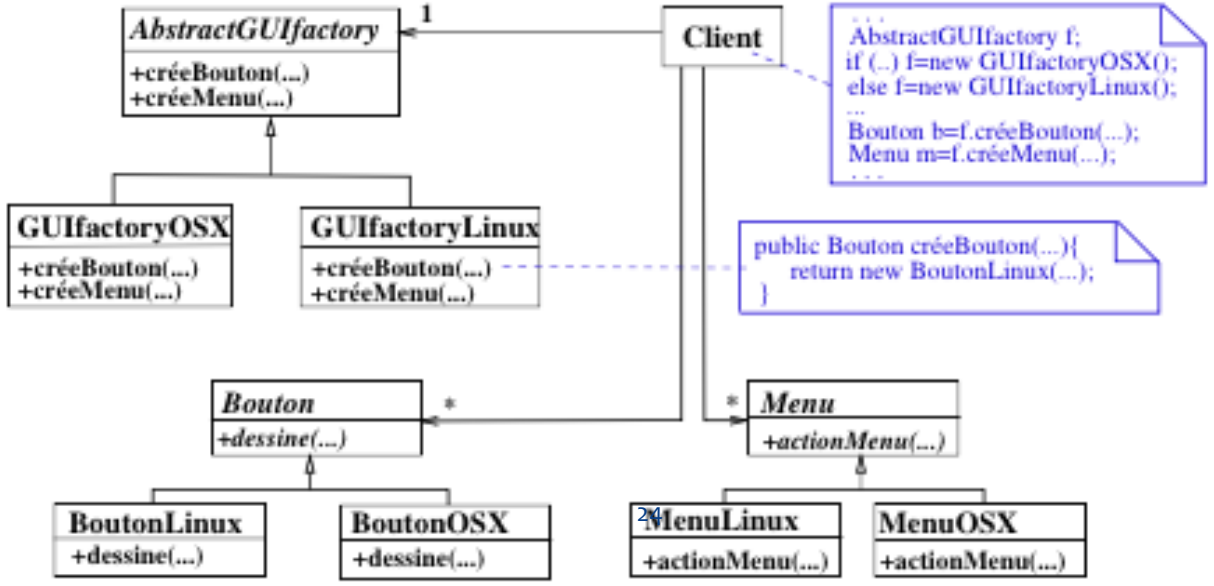
* Fabrique abstraite (abstract factory) :

Problème :

Créer une famille d'objets sans spécifier leurs classes concrètes

Illustration sur un exemple :

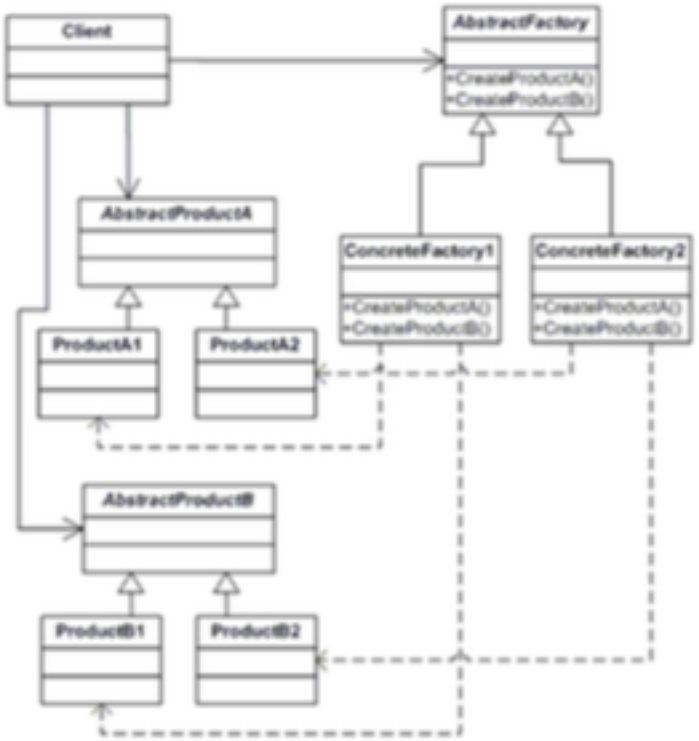
- Créer une interface graphique avec widgets (boutons, menus, ...)
- Point de variation : OS (Linux, OSX, Windows)



Design Patterns

* Factory method (méthode de fabrique)

Solution Générique [Wikipedia] :



Remarques :

- AbstractFactory et AbstractProduct sont généralement des interfaces
  ⇝ Programmer pour des interfaces
- Les méthodes *createProduct. . . ()* sont des *factory methods*

Avantages du pattern :

- Indirection : Isole Client des implémentations des produits
- Protection des variations : Facilite la substitution de familles de produits
- Maintien automatique de la cohérence

Mais l'ajout de nouveaux types de produits est difficile...

Design Patterns

* Singleton :

Problème :

Assurer qu'une classe possède une seule instance et rendre cette instance accessible globalement

Solution générique [Wikipedia] :


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

Exercice :

Utiliser Singleton pour implémenter une classe Factory

Attention :

Parfois considéré comme un anti-pattern... à utiliser avec modération !

Design Patterns

* Iterator (itérateur) :

Problème :

Fournir un accès séquentiel aux éléments d'un agrégat d'objets indépendamment de l'implémentation de l'agrégat (liste, tableau, ...)

Illustration sur un exemple en C++ :

```
class Exemple{
private:
    vector<int> collection;
public:
    int somme(){
        int somme = 0;
        vector<int>::iterator it = collection.begin();
        while (it != collection.end()){
            somme += *it;
            it++;
        };
        return somme;
    }
};
```

Avantage : On peut remplacer `vector<int>` par un autre container sans modifier le code de `somme()`

Design Patterns

* Iterator (itérateur) :

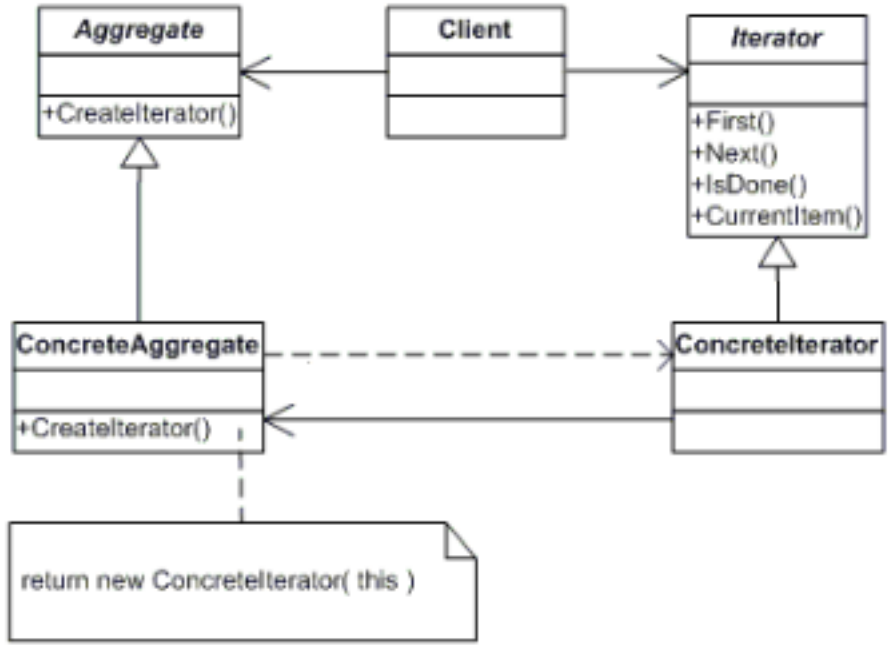
Illustration sur un exemple en Java :

```
public class Exemple {
    private AbstractCollection<Integer> collection;
    public Exemple(){
        collection = new ArrayList<Integer>();
        // ... ajout d'éléments dans collection
    }
    public Integer somme(){
        Integer somme = 0;
        Iterator<Integer> it = collection.iterator();
        while (it.hasNext())
            somme += it.next();
        return somme;
    }
}
```

Design Patterns

* Iterator (itérateur) :

Solution générique :



Avantages :

- Protection des variations : Client est protégé des variations d'Aggregate
- Forte cohésion : Séparation du parcours de l'agrégation
- Possibilité d'avoir plusieurs itérateurs sur un même agrégat en même tps

Design Pattern

* Strategy (stratégie) :

Problème :

Changer dynamiquement le comportement d'un objet

Illustration sur un exemple :

- Dans un jeu vidéo, des personnages combattent des monstres...
  ⇝ méthode `combat(Monstre m)` de la classe `Perso`
  ...et le code de `combat` peut être différent d'un personnage à l'autre
  - Sol. 1 : `combat` contient un cas pour chaque type de combat
  - Sol. 2 : La classe `Perso` est spécialisée en sous-classes qui redéfinissent `combat`

Design Patterns

* Strategy (stratégie) :

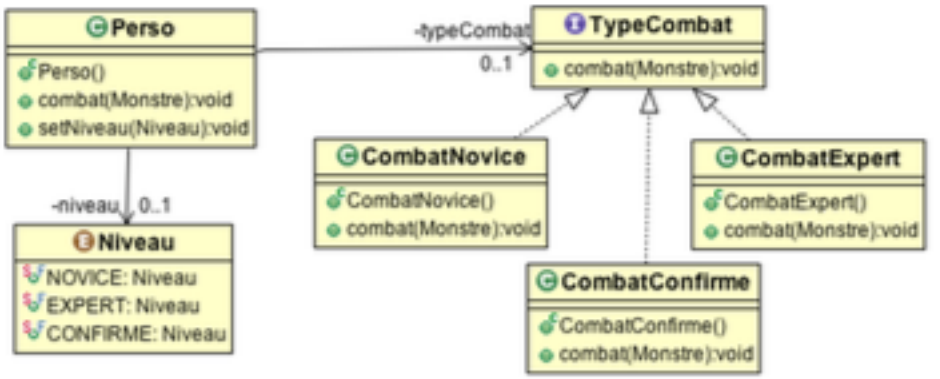
- Représenter ces solutions en UML. Peut-on facilement :
  - Ajouter un nouveau type de combat ?
  - Changer le type de combat d'un personnage ?

Design Patterns

* Strategy :

- Sol. 3 : La classe `Perso` délègue le combat à des classes encapsulant des codes de combat et réalisant toutes une même interface

Diagramme de classes de la solution 3 :



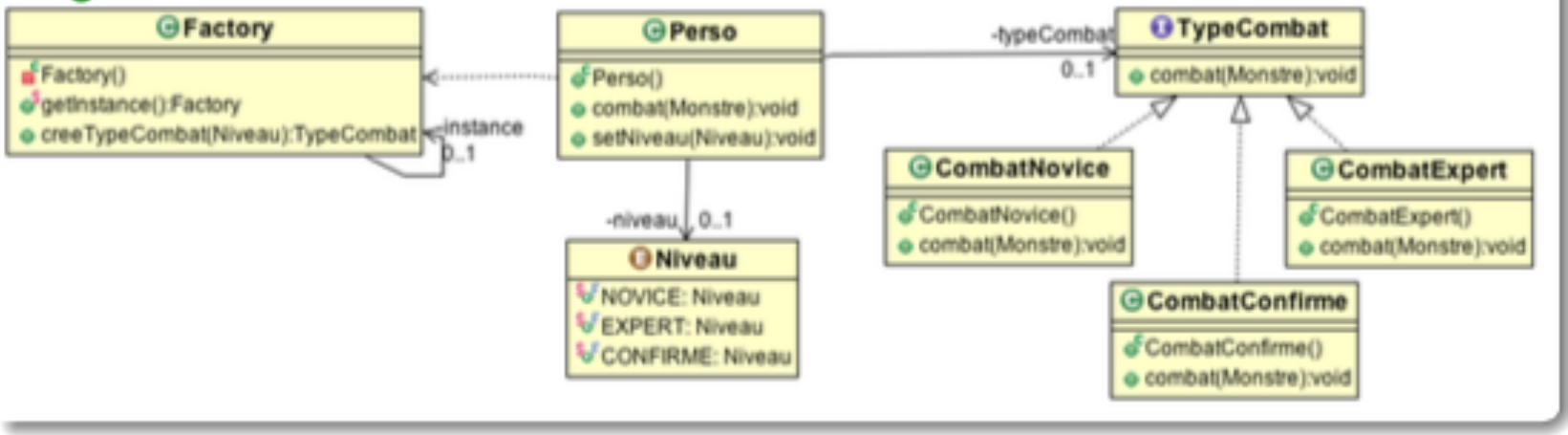
Code Java de la classe `Perso` :

```
public class Perso {
    private TypeCombat typeCombat;
    private Niveau niveau;
    public Perso(){
        niveau = Niveau.NOVICE;
        typeCombat =
    }
    public void combat(Monstre m){
        typeCombat.combat(m);
    }
    public void setNiveau(Niveau niveau) {
        this.niveau = niveau;
        typeCombat =
    }
    // ...Autres méthodes de Perso...
```

Comment créer l'instance de TypeCombat correspondant à niveau ?

Strategy avec Factory :

Diagramme de classes de la solution 3 :



Code Java de la classe `Perso` :

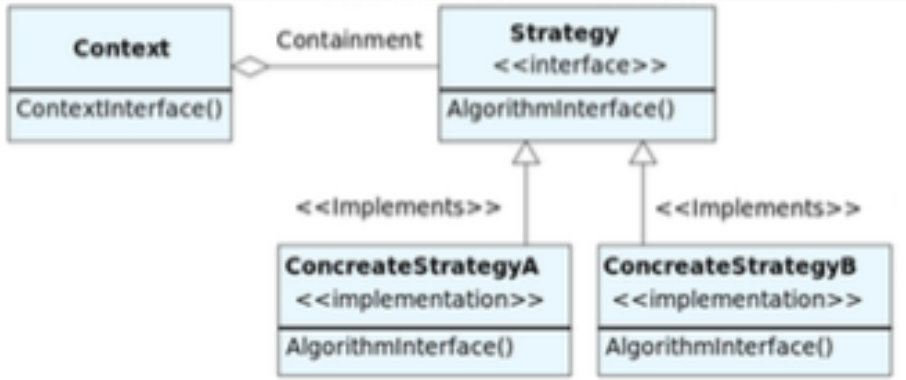
```
public class Perso {
    private TypeCombat typeCombat;
    private Niveau niveau;
    public Perso(){
        niveau = Niveau.NOVICE;
        typeCombat = Factory.getInstance().creeTypeCombat(niveau);
    }
    public void combat(Monstre m){
        typeCombat.combat(m);
    }
    public void setNiveau(Niveau niveau) {
        this.niveau = niveau;
        typeCombat = Factory.getInstance().creeTypeCombat(niveau);
    }
    // ...Autres méthodes de Perso...
```

Design Patterns

* Strategy (stratégie) :

Solution générique :

[Wikipedia]



Remarques :

- Principes de conception orientée objet mobilisés :
  - Indirection : Isole `Context` des implémentations de `Strategy`
    ⇝ Protection des variations
  - Composer au lieu d'hériter : Changer dynamiquement de stratégie
- Passage d'informations de `Context` à `Strategy`
  - en "poussant" : l'info est un param de `AlgorithmInterface()`
  - en "tirant" : le contexte est un param. de `AlgorithmInterface()` qui utilise des getters pour récupérer l'info

Design Patterns

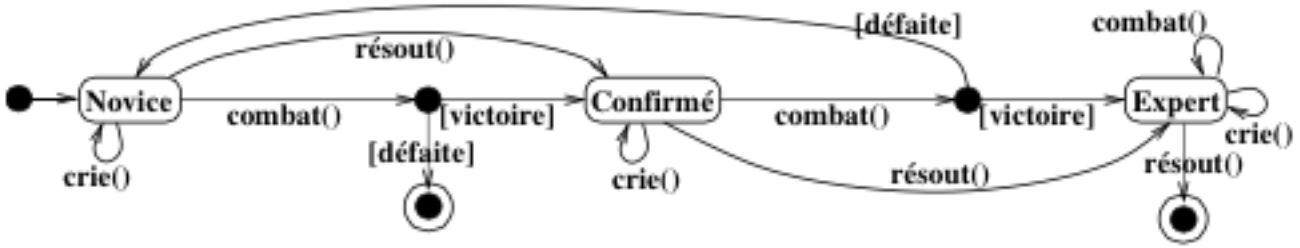
* State (état) :

Problème :

Modifier le comportement d'un objet en fonction de son état

Illustration sur un exemple :

Les personnages d'un jeu peuvent combattre, résoudre des énigmes et crier :



- Sol. 1 : Chaque méthode de `Perso` contient un cas par état

```
public void resout(){
    if (etat == NOVICE){ ...; etat = CONFIRME;}
    else if (etat == CONFIRME){ ...; etat = EXPERT;}
    else { ...; System.exit(0);}
}
```

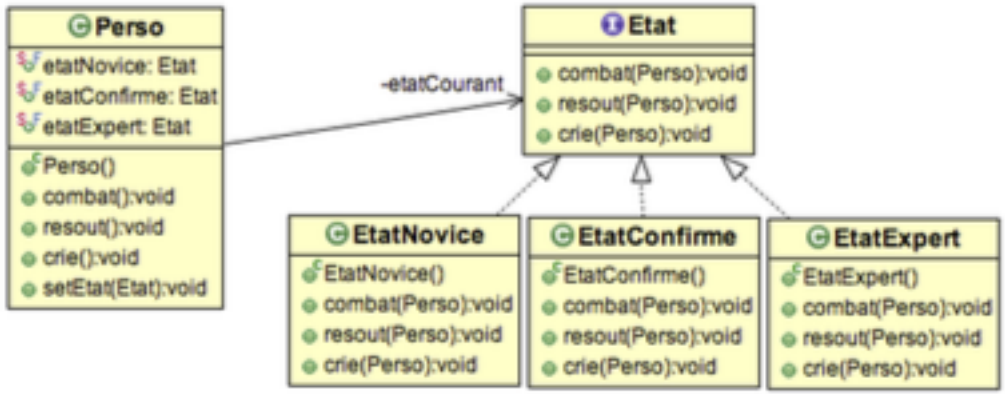

- Quel est le coût de l'ajout d'un nouvel état ou d'une nouvelle action ? 123/1

Design Patterns

* State (état) :

Solution 2 :

Encapsuler les états dans des classes implémentant une même interface




```
public class Perso {
    public static final Etat etatNovice = new EtatNovice();
    public static final Etat etatConfirme = new EtatConfirme();
    public static final Etat etatExpert  = new EtatExpert();
    private Etat etatCourant;
    public Perso(){ etatCourant = etatNovice; }
    public void combat(){ etatCourant.combat(this); }
    public void resout(){ etatCourant.resout(this); }
    public void crie(){ etatCourant.crie(this); }
    public void setEtat(Etat e) { etatCourant = e; }
}
```

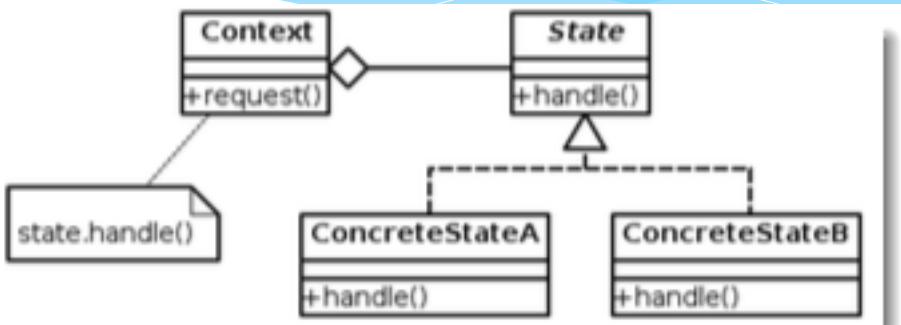
```
public class EtatConfirme implements Etat {
    public void combat(Perso p) {
        // ...code de combat de confirmé
        if (gagnant)
            p.setEtat(Perso.etatExpert);
        else
            p.setEtat(Perso.etatNovice);
    }
    // ...
```

Design Patterns

* State (état) :

Solution générique :

[Wikipedia]



Remarques :

- Ajout d'un nouvel état facile...
  ...mais ajout d'une nouvelle action plus compliqué
- Si `ConcreteState` ne mémorise pas d'information interne
  Alors les états peuvent être des attributs statiques de `Context`
  Sinon il faut une instance de `ConcreteState` par instance de `Context`
  ⇝ Peut devenir coûteux en mémoire !
- Point commun avec Strategy : utilise la délégation pour modifier dynamiquement le comportement des instances de `Context`, comme si elles changeaient de classes

Design Patterns

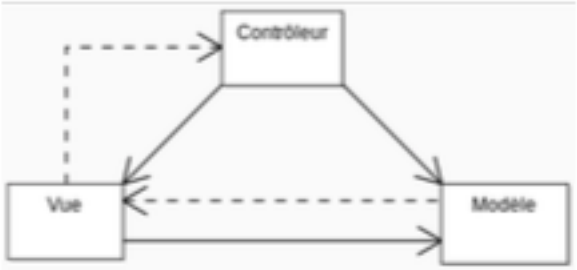
* Observer (Observeur aka Publish/Subscribe)

Problème :

Faire savoir à un ensemble d'objets (observateurs/abonnés) qu'un autre objet (observable/publieur) a été modifié

Illustration sur l'exemple du jeu vidéo :

La représentation (vue) des personnages dépend de leur niveau...
...et on peut avoir plusieurs vues (graphique, sonore, textuelle, etc)
...et on veut se protéger des évolutions et variations sur ces vues
⇝ Utilisation du patron architectural Model-View-Controller (MVC)



- Flèche pleine = dépendance
- Pointillés = événements

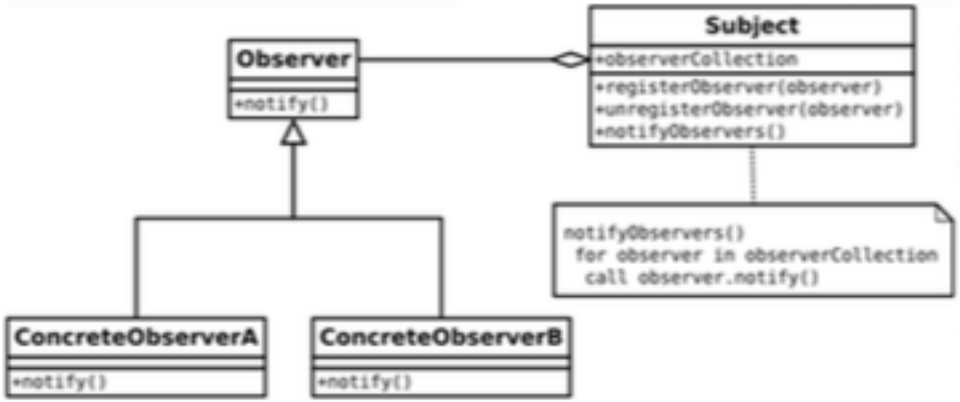
- Modèle : traite les données
- Vue : affiche les données, reçoit les evt clavier/souris et les envoie au contrôleur
- Contrôleur : analyse les evt clavier/souris et active le modèle et/ou la vue en conséquence

Modèle est observable. Vue est observateur

Design Patterns

* Observer (Observeur aka Publish/Subscribe)

Solution générique [Wikipedia] :



Remarques :

- Faible couplage entre `ConcreteObserver` et `Subject`
- Les données de Subject peuvent être “poussées” (dans `notify`) ou “tirées” (avec des `getters`)
- Se retrouve dans de nombreuses API Java
  ⇝ “Listeners” de l’API Swing pour observer le clavier, la souris, ...

Design Patterns

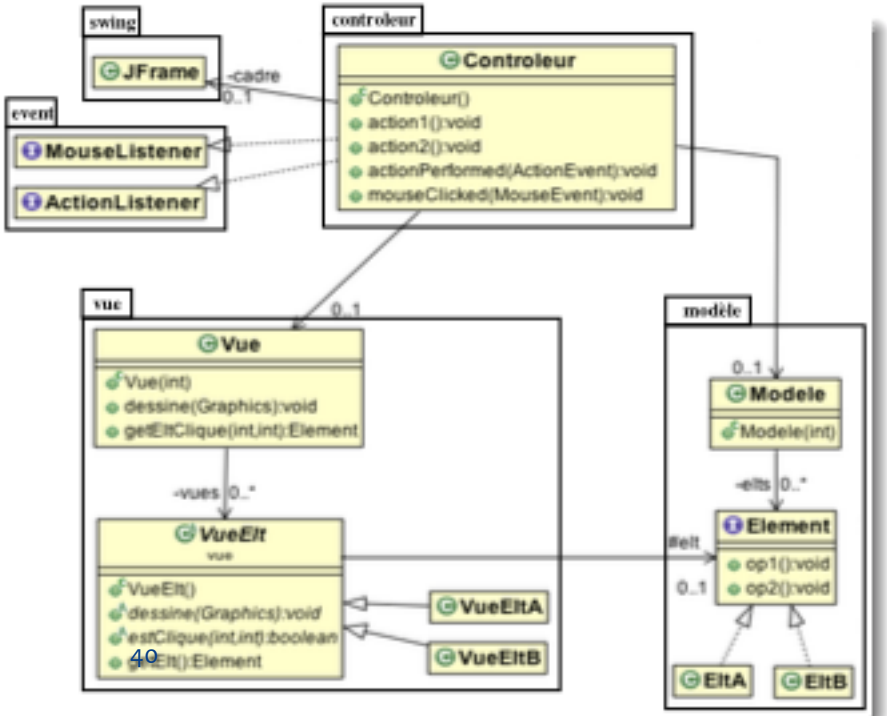
* Command (commande) :

Problème :

Découpler la réception d'une requête de son exécution

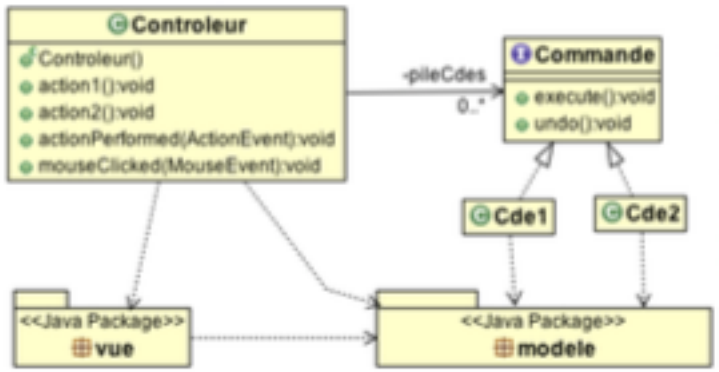
Illustration / modèle MVC :

- Le contrôleur reçoit les événements utilisateur (`actionPerformed` et `mouseClicked`) et appelle en conséquence des méthodes (`action1` et `action2`) qui activent le modèle et la vue
- On veut garder un historique pour pouvoir annuler les dernières actions



Design Patterns

* Command (commande) :
  - Encapsuler les commandes dans des objets contenant les informations permettant de les exécuter/annuler
  - Stocker les commandes dans une pile



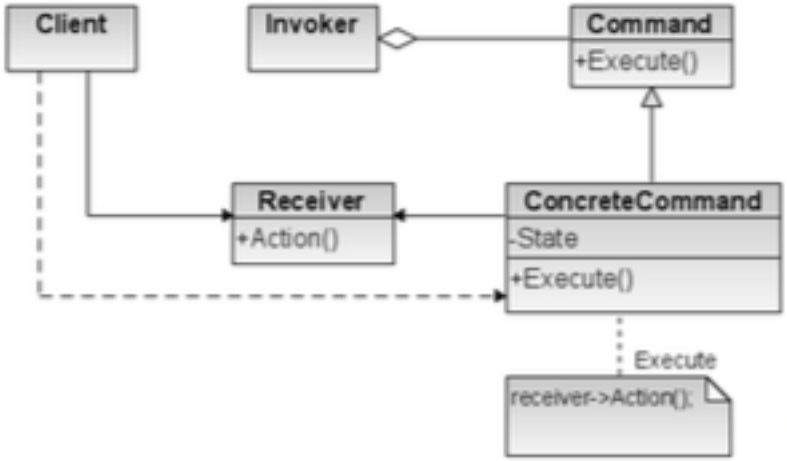

```
public class Controleur
implements ActionListener, MouseListener{
    private Modele modele;
    private Vue vue;
    private Stack<Commande> pileCdes;
    public Controleur() {
        modele = new Modele(paramsModele);
        vue = new Vue(paramsVue);
        pileCdes = new Stack<Commande>();
    }
    public void action1() {
        // ...
        Commande cde = new Cde1(paramsCde);
        pileCdes.push(cde);
        cde.execute();
    }
    public void undo() {
        if (!pileCdes.empty()){
            Commande c = pileCdes.pop();
            c.undo();
        }
        else System.out.println("Pas de cde à annuler");
    }
```

Design Patterns

* Command (commande) :

Solution générique :

- `Client` crée les instances de `ConcreteCommand`
- `Invoker` demande l'exécution des commandes
- `ConcreteCommande` délègue l'exécution à `Receiver`



Remarques :

- Découple la réception d'une requête de son exécution
- Les rôles de `Client` et `Invoker` peuvent être joués par une même classe (par exemple le contrôleur)
- Permet la journalisation des requêtes pour reprise sur incident
- Permet d'annuler ou ré-exécuter des requêtes (undo/redo)

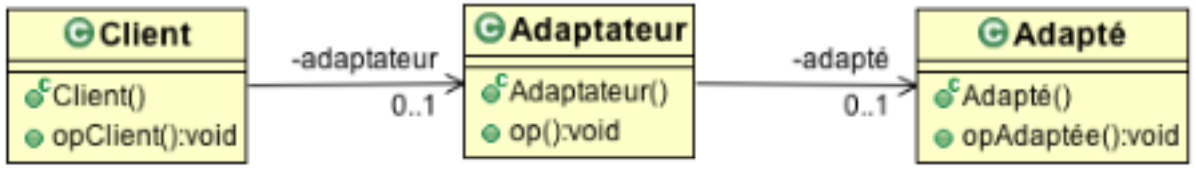
Design Patterns

* Adapter (adapteur)

Problème :

Fournir une interface stable (Adaptateur) à un composant dont l'interface peut varier (Adapté)

Solution générique :



⇝ Application des principes "indirection" et "protection des variations"

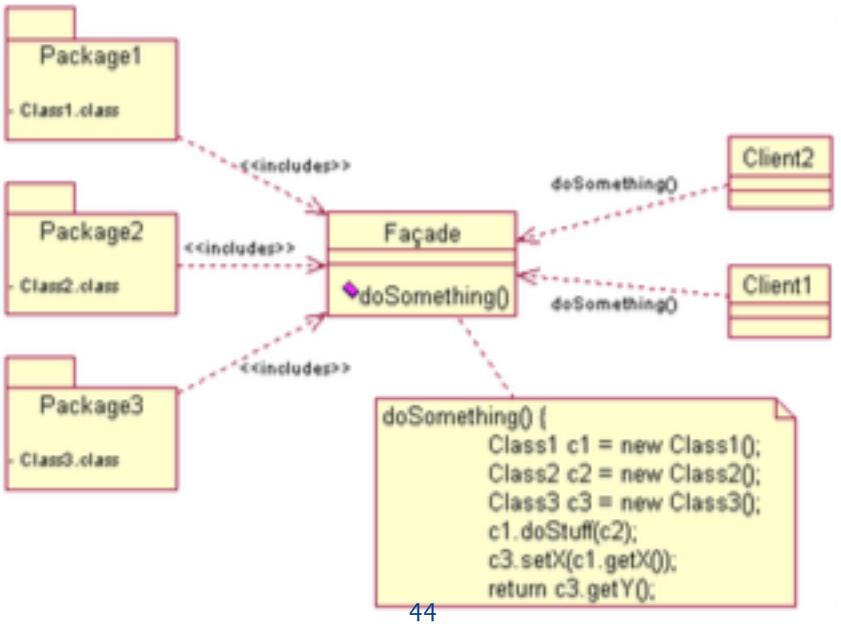
Design Patterns

* Facade (façade) :

Problème :

Fournir une interface simplifiée (Facade)

Solution générique [Wikipedia] :



⇝ Application des principes “indirection” et “protection des variations”

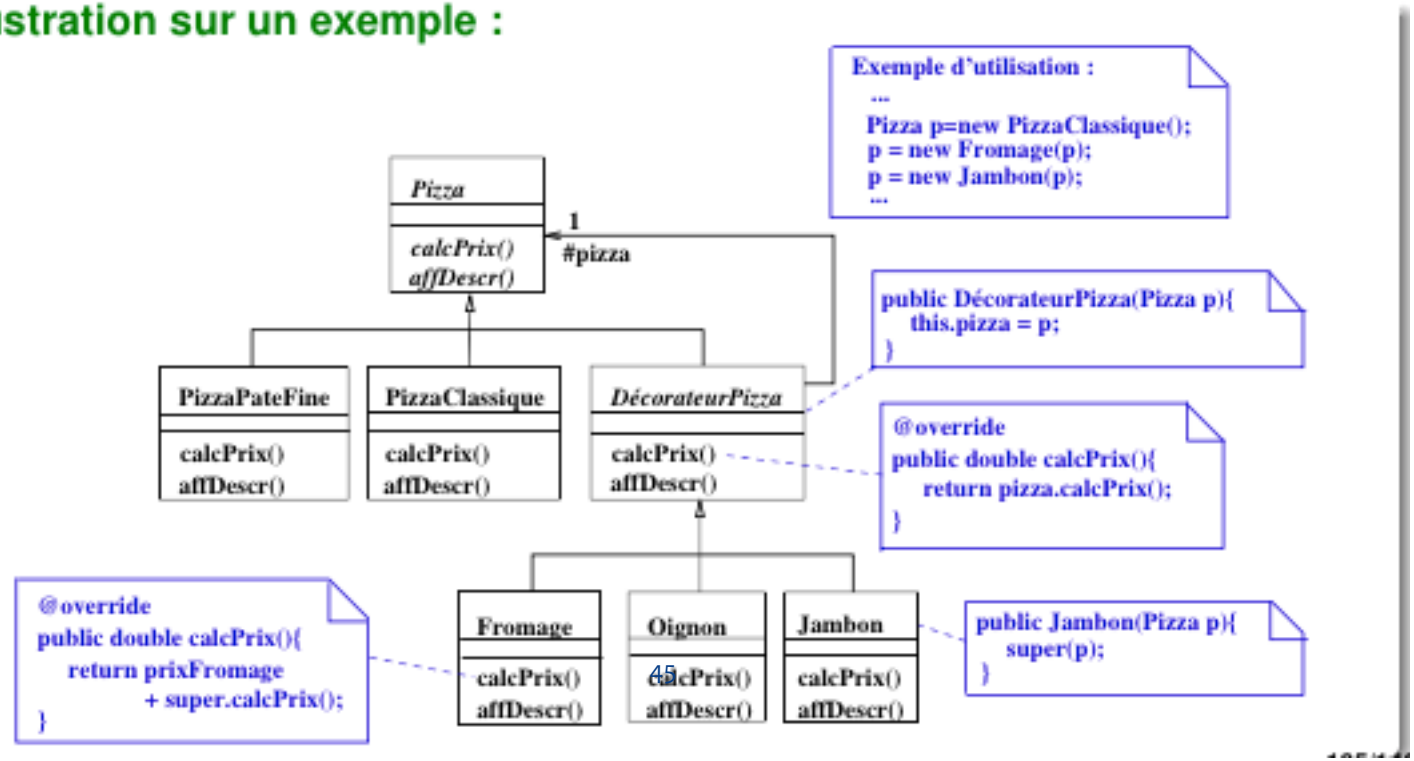
Design Patterns

* Decorator (décorateur) :

Problème :

Attacher dynamiquement des responsabilités supplémentaires à un objet

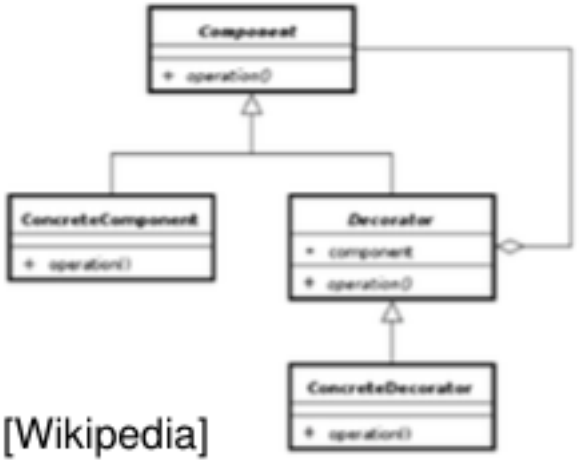
Illustration sur un exemple :



Design Patterns

* Decorator (décorateur) :

Solution générique :



[Wikipedia]

Remarques :

- Composer au lieu d'hériter : Ajout dynamique de responsabilités à ConcreteComponent sans le modifier
- n décors $\Rightarrow 2^n$ combinaisons
- **Inconvénient** : Peut générer de nombreux petits objets "enveloppes"

Utilisation pour décorer les classes d'entrée/sortie en Java :

- Component : InputStream, OutputStream
- ConcreteComponent : FileInputStream, ByteArrayInputStream, ...
- Decorator : FilterInputStream, FilterOutputStream
- ConcreteDecorator : BufferedInputStream, CheckedInputStream, ...

Design Patterns

* Adapter, Facade, Decorator :

Points communs :

- Indirection ⇝ Enveloppe (wrapper)
- Protection des variations

Différences :

- Adapter : Convertit une interface en une autre (attendue par un Client)
- Facade : Fournit une interface simplifiée
- Decorator : Ajoute dynamiquement des responsabilités aux méthodes d'une interface sans la modifier

Design Patterns

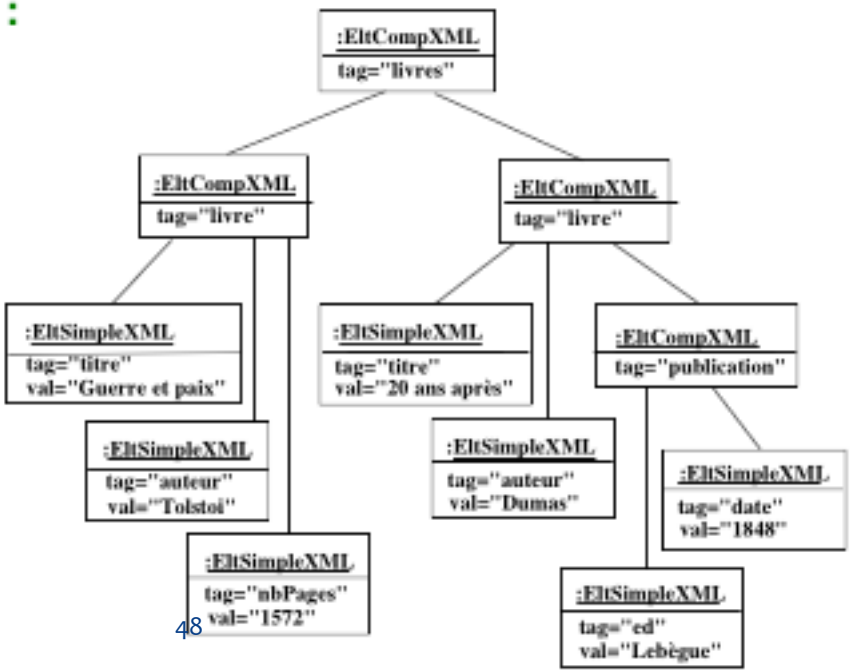
* Composite :

Problème :

Représenter des hiérarchies composant/composé et traiter de façon uniforme les composants et les composés

Illustration sur un exemple :

```
< ?xml version="1.0" ?>
<livres>
    <livre>
        <titre>Guerre et paix</titre>
        <auteur>Tolstoï</auteur>
        <nbPages>1572</nbPages>
    </livre>
    <livre>
        <titre>20 ans après</titre>
        <auteur>Dumas</auteur>
        <publication>
            <ed>Lebègue</ed>
            <date>1848</date>
        </publication>
    </livre>
</livres>
```

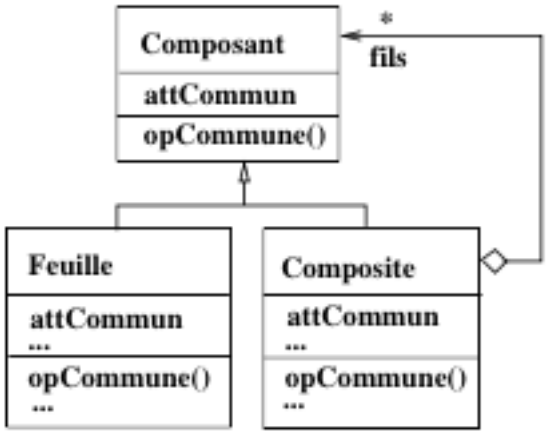


Comment compter le nombre de tags ?

Design Patterns

* Composite :

Solution générique :



Exercices :

- Définir les opérations permettant de :
  - Compter le nombre de fils d'un composant
  - Compter le nombre de descendants d'un composant
  - Ajouter un fils à un composant
- Comment accéder séquentiellement aux fils d'un Composite ?
- Comment accéder séquentiellement aux descendants d'un Composite ?