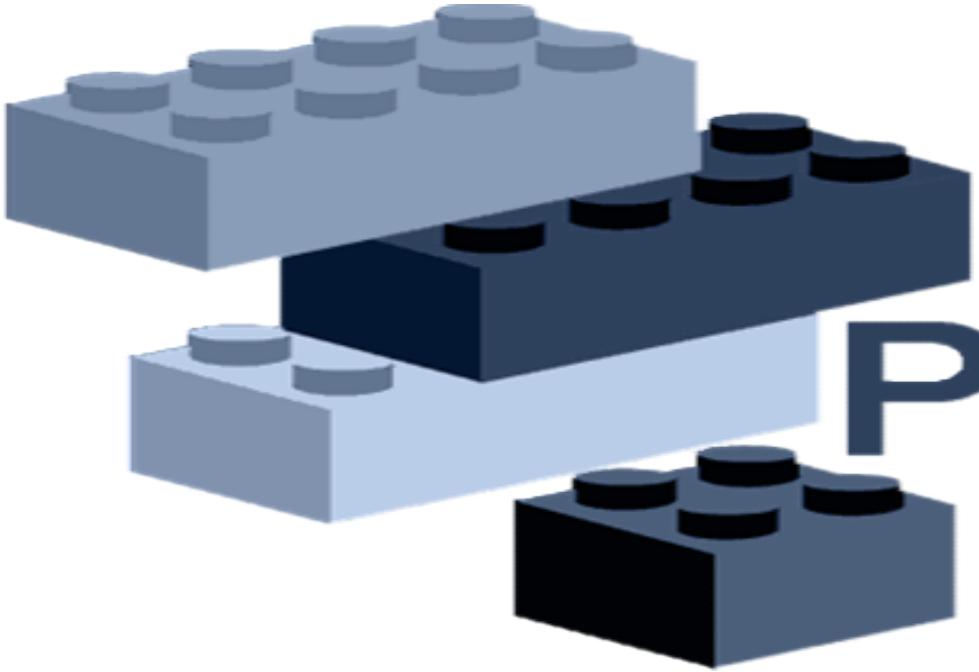
ECOLE MAROCAINE DES SCIENCES DE L'INGENIEUR
Membre de HONORIS UNITED UNIVERSITIES

Conception Orientée Objet

3IIR



Design Patterns

Pr. Khalid SRAIDI

A.U 2022/2023

Introduction

- **Les patrons de conception** (**Design Patterns**) constituent des **réponses** éprouvées à des **problèmes** de conception récurrents.
- Un patron de conception (Design Pattern) n'est pas le code final, mais une méthode modèle permettant de savoir comment aborder un problème dans un nouveau scénario pendant la phase de développement du logiciel.
- Ils sont des logiciels orientés objet réutilisables qui répondent aux défis les plus courants du développement de logiciels.
- Cela décrit à la fois la description d'une solution et son utilisation pour résoudre un problème particulier.
- Chaque **Patron** est une sorte de plan ou de schéma que vous pouvez personnaliser afin de résoudre un problème récurrent dans votre code.

Objectifs

- **Modularité**
  - Facilité de gestion (technologie objet)
- **Cohésion (Voir Annexe A)**
  - Degré avec lequel les tâches réalisées par un seul modules sont fonctionnellement reliées.
  - Une forte cohésion est une bonne qualité.
- **Couplage (Voir Annexe B)**
  - Degré d'interaction entre les modules dans le système.
  - Un couplage ''lâche'' est une bonne qualité.
- **Réutilisabilité**
  - Bibliothèques, frameworks (cadres)

Définitions: Patron

« Un patron décrit à la fois un **problème** qui se produit très fréquemment dans l'environnement et l'architecture de la **solution** à ce problème de telle façon que l'on puisse **utiliser** cette solution des milliers de fois sans jamais l'**adapter** deux fois de la même manière. »

‘’***C. Alexander*** ‘’

*« Les patrons de conception vous aident à apprendre des succès des autres plutôt que de vos propres échecs. » ‘’**Mark Johnson**’’*

Catégories de Patterns

- **Patrons de Conception (Design Patterns)**
  – caractéristiques clés d'une structure de conception commune à plusieurs applications.
- **Patrons d'architecture** (**Architectural Patterns)**
  – schémas d'organisation structurelle de logiciels (pipes, filters, brokers,MVC, …)
- **Idioms ou coding patterns**
  – solution liée à un langage particulier.
- **Anti-patterns**
  – mauvaise solution ou comment sortir d'une mauvaise solution.
- **Organizational patterns**
  – Schémas d'organisation de tout ce qui entoure le développement d'un logiciel

Présentation d'un Design Pattern

- **Nom du pattern**
  – utilisé pour décrire le pattern, ses solutions et les conséquences en un mot ou deux.
- **Problème**
  – description des problèmes à résoudre son contexte.
- **Solution**
  – description des éléments (objets, relations, responsabilités, collaboration) permettant de concevoir la solution au problème ; utilisation de diagrammes de classes, de séquences, … vision statique ET dynamique de la solution
- **Conséquences**
  – description des résultats (effets induits) de l 'application du pattern sur le système (effets positifs ET négatifs)

Portée des Design Patterns

On veut dire par portée, l'étendu du champ d'action des Design Patterns.

- Portée de Classe
  - Focalisation sur les relations entre classes et leurs sous-classes.
  - Réutilisation par **héritage**.

- Portée d'Instance
  - Focalisation sur les relations entre les objets
  - Réutilisation par **composition**.

Organisation des patrons de conception

La bande des quatre (GoF,Gang Of Four),(Gamma, Helm, Johnson, Vlissides) présente 23 patrons de conception,

		Catégorie		
		Création	**Structure**	**Comportement**
Portée	**Classe**	Factory Method	Adapter	Interpreter
				Template Method
	Objet	Abstract Factory	Adapter	Chain of Responsibility
		Builder	Bridge	Command
		Prototype	Composite	Iterator
		Singleton	Decorator	Mediator
			Facade	Memento
			Flyweight	Observer
			Proxy	State
				Strategy
				Visitor

Catégories de Design Patterns

1. Les Patrons de Création:

- Description de la manière dont un objet ou un ensemble d'objets peuvent être créés, initialisés, et configurés.
- Isolation du code relatif à la création, à l'initialisation afin de rendre l'application indépendante de ces aspects.

Problème 1

Instancier une classe, une seule et unique fois :

Garantir qu'une classe aura **une seule instance** et fournir **un accès global** à cette instance.

Exemple :

- Le cas de **l'implémentation** d'une classe servant de **pilote pour un périphérique** ou une classe de **connexion à une base de données**.
- Instancier **2 fois** une classe servant de pilote à une imprimante ou de connexion à une base des données provoquerait **une surcharge** inutile du système et des **comportements incohérents**.

Catégories de Design Patterns

2. Les Patrons Structurels

- Description de la manière dont doivent être connectés des objets de l'application afin de rendre ces connections **indépendantes** des évolutions futures de l'application.
- **Découplage** de l'interface et de l'implémentation de classes et d'objets.

Problème 2

Représenter des hiérarchies composant/composé :

Modéliser le **système de gestion de fichiers** qui regroupent les éléments :

- des **fichiers**,
- des **raccourcis**,
- des **répertoires**

Tous ces éléments **sont contenus dans des répertoires**

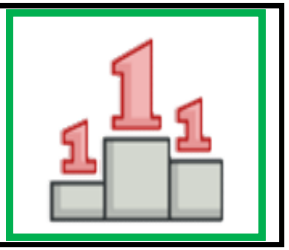
Catégories de Design Patterns

3. Les Patrons Comportementaux

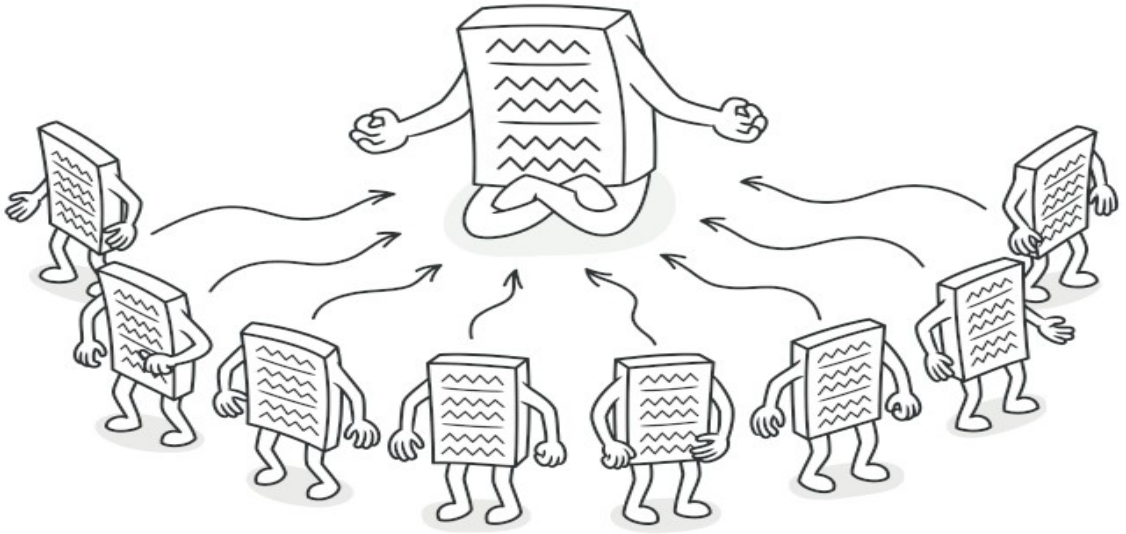
- Les patrons comportementaux s'occupent des algorithmes et de la répartition des responsabilités entre les objets.
- Gestion des interactions dynamiques entre des classes et des objets.

https://codepen.io/lorp/pen/RMmRgB

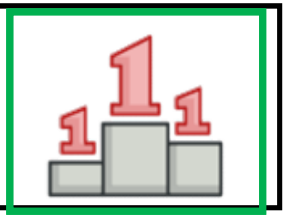
1.1 Design Patterns de création
«Singleton »



1.1 Intention :Singleton est un patron de conception de création qui a pour objectif de *garantir qu'une classe n'a **qu'une instance** et fournir un point **d'accès global** à cette classe*.



1.1 Design Patterns de création
«Singleton »



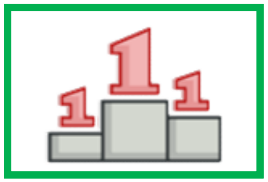
1.2. Problème:

- Divers clients doivent se référer à une même chose et on veut être sûr qu'il n'existe qu'une seule instance.
- Si on ne garantit pas l'unicité, il peut y avoir de graves problèmes de fonctionnement (eg, incohérences, écrasement de données).

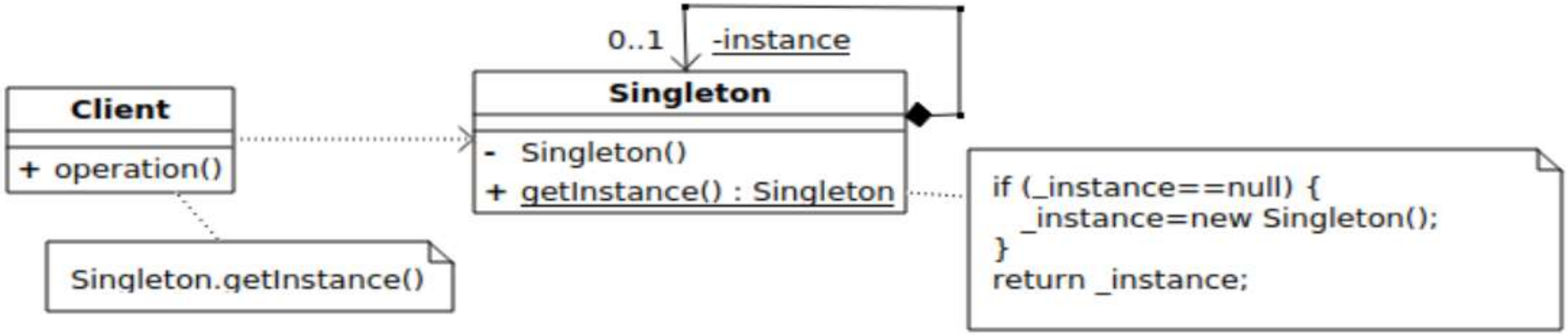
1.3. Solution:

- S'assurer qu'il n'existe qu'une seule instance en contrôlant le constructeur.

1.1 Design Patterns de création
«Singleton »



1.4 Structure:



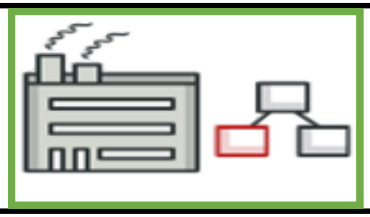
1.5 Constituants:

• **Singleton** : construit sa propre instance **(-)** unique et définit une opération getInstance() qui donne l'accès (global **(+)**) à son unique instance.

1.6 Collaborations:

Les clients accèdent à l'instance uniquement par l'intermédiaire de l'opération getInstance() de la classe Singleton.

1.2 Design Patterns de création
Factory



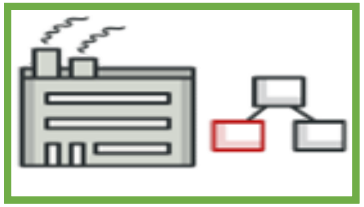
- **Intention: Factory (Fabrique)** est un patron de conception de création qui définit une interface pour créer des objets dans une classe mère, mais délègue le choix des types d'objets à créer aux sous-classes.
- **Problème**

✓Imaginez que vous êtes en train de créer une application de gestion logistique.

✓La première version de votre application ne propose que le transport par camion.

✓la majeure partie de votre code est donc située dans la classe Camion.

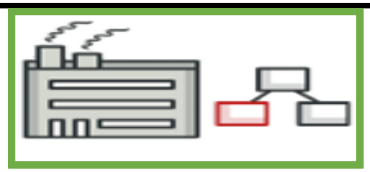
1.2 Design Patterns de création
Factory



Description : Au bout d'un certain temps, votre application devient populaire et de nombreuses entreprises de transport maritime vous demandent tous les jours d'ajouter la gestion de la logistique maritime dans l'application.

L'ajout d'une nouvelle classe au programme ne s'avère pas si simple que cela si le reste du code est déjà couplé aux classes existantes.

1.2 Design Patterns de création
Factory

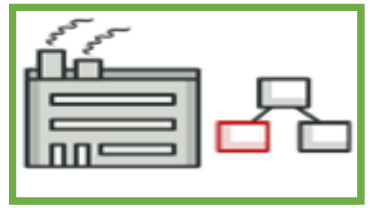


Description:

La majeure partie est actuellement couplée à la classe Camion. Pour pouvoir ajouter des Bateaux dans l'application, il faudrait revoir la base du code. De plus, si vous décidez plus tard d'ajouter un autre type de transport dans l'application, il faudra effectuer à nouveau ces changements.

Par conséquent, vous allez vous retrouver avec du code pas très propre, rempli de conditions qui modifient le comportement du programme en fonction de la classe des objets de transport.

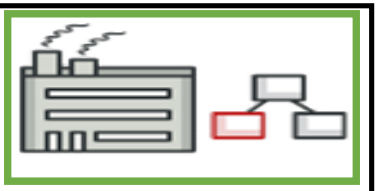
1.2 Design Patterns de création
Factory



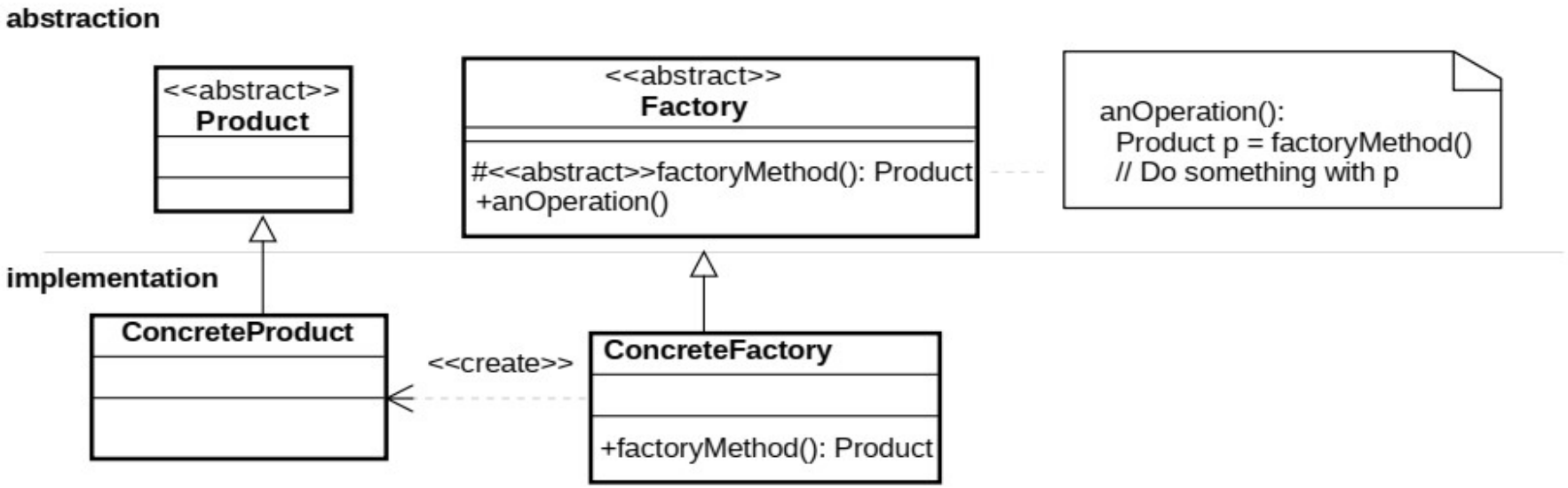
Solution: Le patron de conception fabrique vous propose de remplacer les appels directs au constructeur de l'objet (à l'aide de l'opérateur New) en appelant une méthode *fabrique* spéciale.

les objets sont toujours créés avec l'opérateur New, mais l'appel se fait à l'intérieur de la méthode fabrique. Les objets qu'elle retourne sont souvent appelés *produits.*

1.2 Design Patterns de création
Factory



Structure:



Participants

- **Product** : définit l'interface des objets créés par la fabrique.
- **ConcreteProduct** : implémente l'interface Product avec un produit concret.
- **Factory** : déclare l'interface de la fabrique : celle-ci renvoie un type de produit.

ConcreteFactory : surcharge la fabrique pour renvoyer une instance d'un produit concret.

2. Design Patterns Structurels
« Composite »

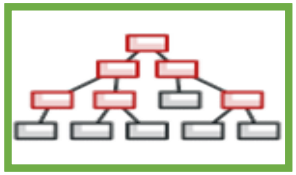
2.1 Intention: *Créer des objets composites avec une structure arborescente pour permettre au client de traiter de la même façon des objets individuels et des groupements de ces objets.*

Solution

• Définir une classe abstraite qui spécifie le comportement commun à tous les composés et composants.

• La relation de composition lie un composé à tout objet du type de la classe abstraite.

2. Design Patterns Structurels
«Composite»



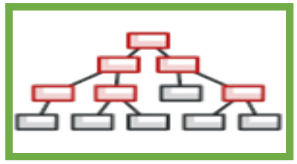
2.2 Exemple :

Prenons les deux objets suivants: Produits et Boites.

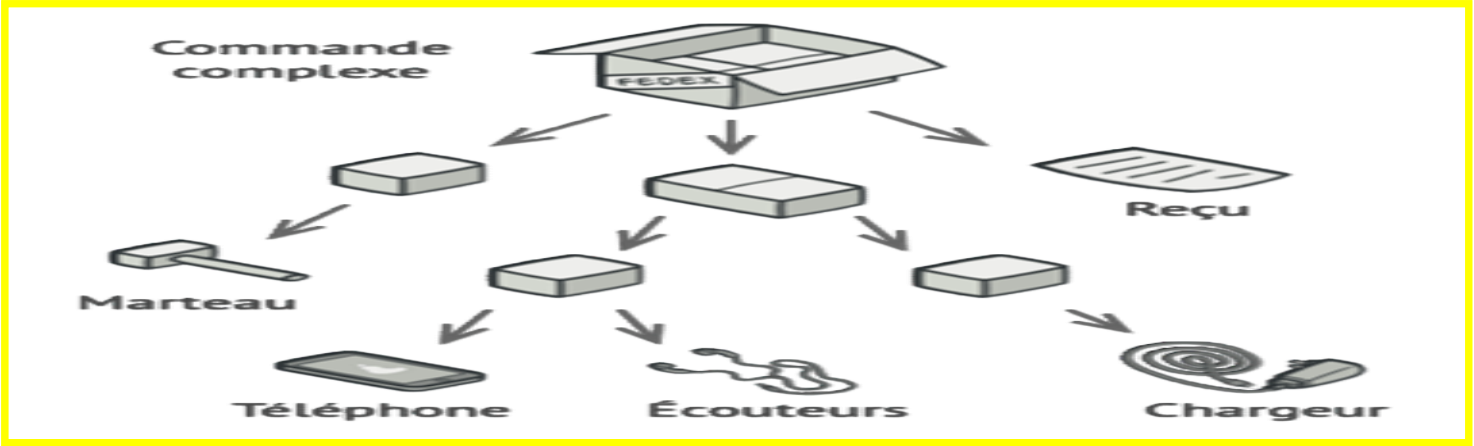
Une boîte peut contenir plusieurs produits ainsi qu'un certain nombre de boîtes plus petites. Ces petites boîtes peuvent également contenir quelques produits ou même d'autres boîtes encore plus petites, et ainsi de suite.

Vous décidez de mettre au point un système de commandes qui utilise ces classes. Les commandes peuvent être composées de produits simples sans emballage, d'autres boîtes remplies de produits... et d'autres boîtes. Comment allez-vous déterminer le coût total d'une telle commande ?

2. Design Patterns Structurels
« Composite »

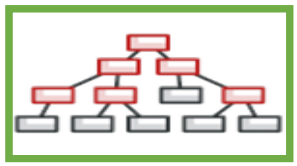


Vous décidez de mettre au point un système de commandes qui utilise ces classes. Les commandes peuvent être composées de produits simples sans emballage, d'autres boîtes remplies de produits... et d'autres boîtes. Comment allez-vous déterminer le coût total d'une telle commande ?



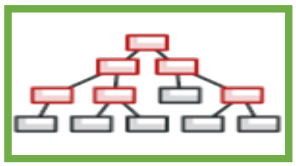
Une commande peut contenir divers produits empaquetés à l'intérieur de boîtes, elles-mêmes rangées dans de plus grosses boîtes, etc. La structure complète ressemble à un arbre inversé

2. Design Patterns Structurels
« Composite »



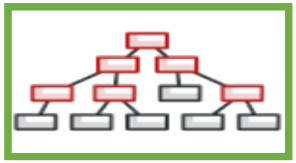
- ✓Vous pouvez tenter l'approche directe qui consiste à déballer toutes les boîtes, prendre chaque produit et en faire la somme pour obtenir le total.
- ✓ Ce mode de calcul peut facilement se mettre en place dans le monde réel mais dans un programme, ce n'est pas aussi simple que de créer une boucle.
- ✓ Il faut connaître à l'avance la classe des produits et des boites que l'on parcourt, le niveau d'imbrication des boîtes ainsi que d'autres détails.
- ✓Tout ceci rend l'approche directe assez compliquée et même parfois impossible.

2. Design Patterns Structurels
« Composite »



Problématique

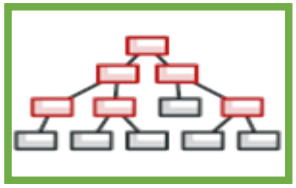
2. Design Patterns Structurels
« Composite »



2.3 Problématique

- Nécessité de représenter des hiérarchies de l'individu à l'ensemble.
- Ne pas de se soucier de la différence entre combinaisons d'objets et constituants.

2. Design Patterns Structurels
« Composite »

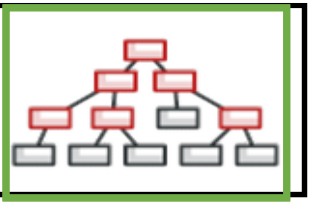


Solution

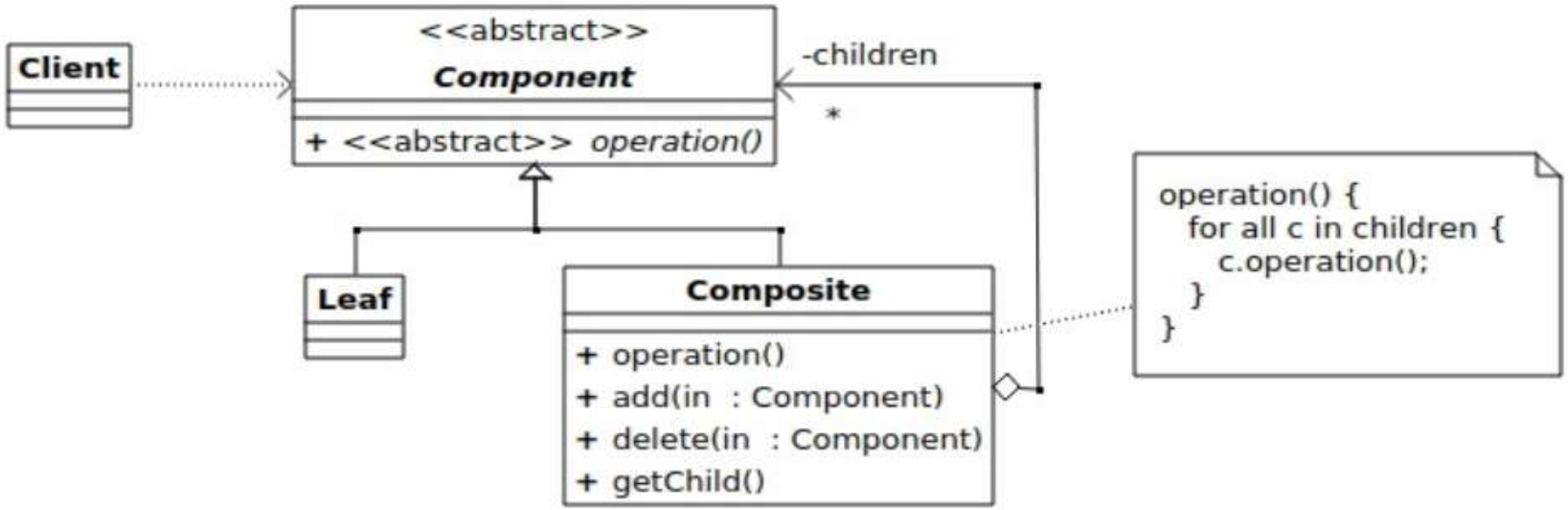
Le patron de conception composite vous propose de manipuler les produits et boites à l'aide d'une interface qui déclare une méthode de calcul du prix total.

- ✓Comment cette méthode peut-elle fonctionner ? Pour un produit, on retourne simplement son prix.
- ✓Pour une boîte, on parcourt chacun de ses objets, on leur demande leur prix, puis on retourne un total pour la boîte.
- ✓Si l'un de ces objets est une boîte plus petite, cette dernière va aussi parcourir son propre contenu et ainsi de suite, jusqu'à ce que tous les prix aient été calculés.
- ✓Une boîte peut même ajouter des frais supplémentaires, comme le prix de l'emballage.

2. Design Patterns Structurels
« Composite »



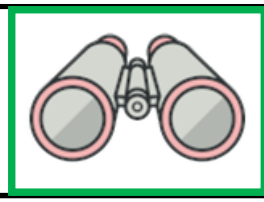
Structure:



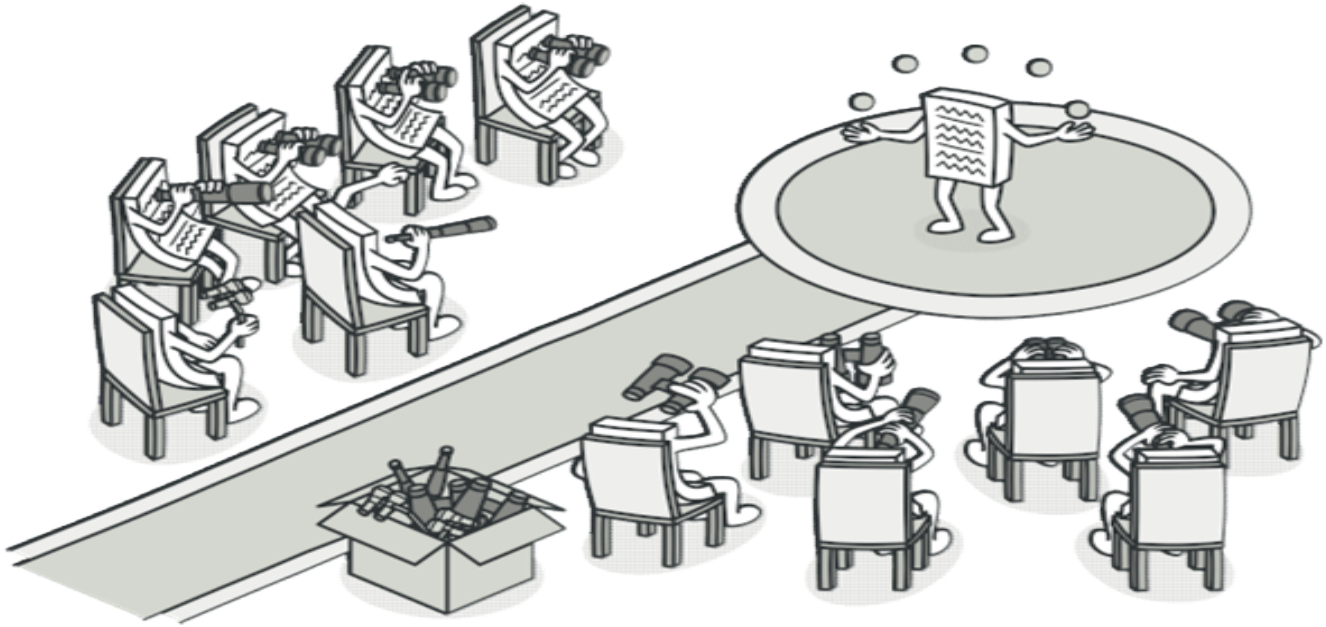
Constituants:

• **Component** : déclare l'interface des objets entrants dans la composition, implante le comportement commun et déclare une interface pour accéder à ses composants enfants.
• **Leaf** : représente les objets terminaux dans la composition et définit le comportement des objets primitifs.
• **Composite** : définit le comportement des objets composés et stocke les composants enfants sous la forme de listes.

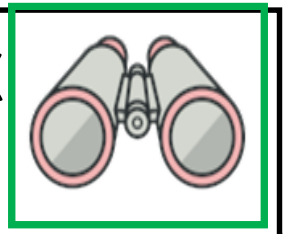
3.Design Patterns Comportementaux
« Observateur »



3.1. Intention: L'**Observateur** (Observer) est un patron de conception comportemental qui *définit une dépendance "**1 à plusieurs**" et mets en place un mécanisme de souscription entre un objet et ses multiples observateurs de manière à ce que les observateurs soient automatiquement notifiés d'un changement d'état de l'objet qu'ils observent.*

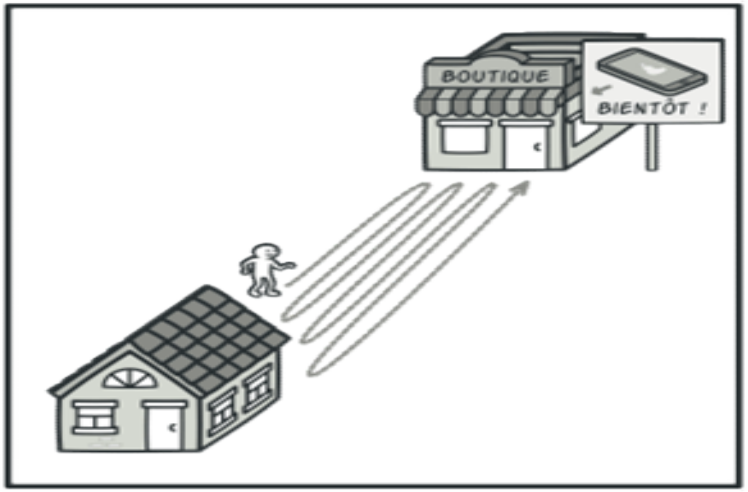


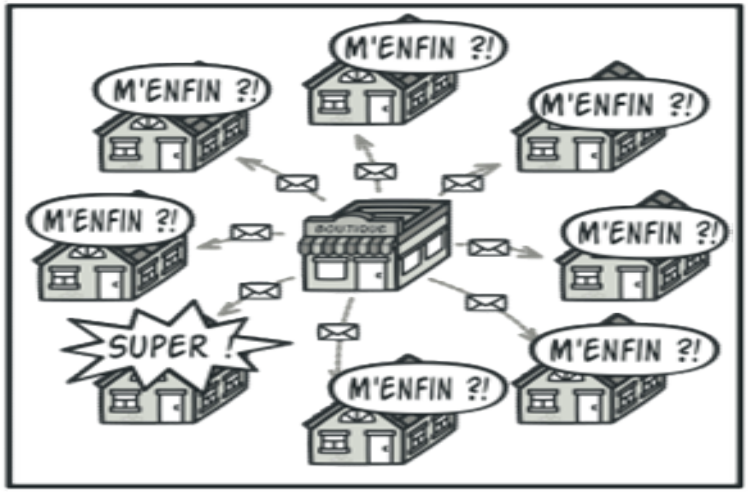
3. Design Patterns Comportementaux
« Observateur »



3.2.1 Exemple:

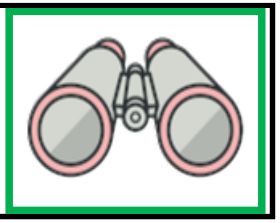
- ✓Imaginez que vous avez deux types d'objets : un client et un magazin.
- ✓Le client s'intéresse à une marque spécifique d'un produit (disons que c'est un nouveau modèle d'iPhone) qui sera bientôt disponible dans la boutique.
- ✓Le client pourrait se rendre sur place tous les jours et vérifier la disponibilité du produit. Mais comme le produit n'est pas encore prêt, ses allées et venues seraient inutiles.





Se rendre au magasin *ou* *envoyer du spam*

3. Design Patterns Comportementaux
« Observateur »



✓À la place, le magasin pourrait envoyer des tonnes d'e-mails (ce qui peut être vu comme du spam) à leurs clients chaque fois qu'un nouveau produit est disponible.

✓ **Une solution** qui économiserait bien des voyages aux clients. En contrepartie, le magasin **risque** de se mettre à dos (i.e. qui ne va pas dans le même sens) ceux qui ne sont pas intéressés par les nouveaux produits.

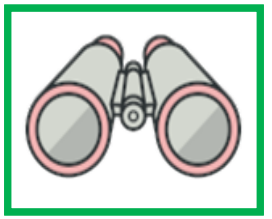
✓Nous nous retrouvons donc, dans une situation **conflictuelle**;

a) Soit les clients perdent leur temps à venir vérifier la disponibilité des produits,

b) Soit le magasin gâche des ressources pour prévenir des clients qui ne sont pas concernés.

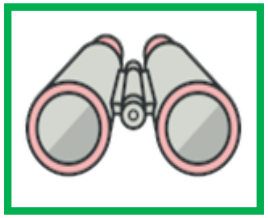
3. Design Patterns Comportementaux

« Observateur »



Problématique

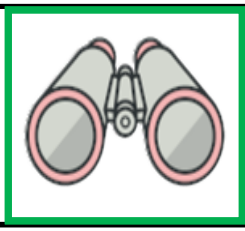
3. Design Patterns Comportementaux
« Observateur »



3.2. Problématique

- Informer une liste variable d'objets qu'un événement a eu lieu.
- **Ce qui varie** : la liste des objets intéressés par des événements d'un objet de référence.

3.Design Patterns Comportementaux
« Observateur »



3.3 Solution:

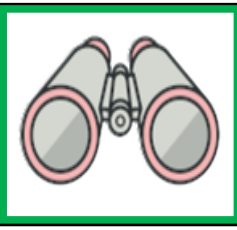
✓L'objet que l'on veut suivre est en général appelé *sujet*, mais comme il va envoyer des notifications pour prévenir les autres objets dès qu'il est modifié, nous l'appellerons ***diffuseur*** (publisher).

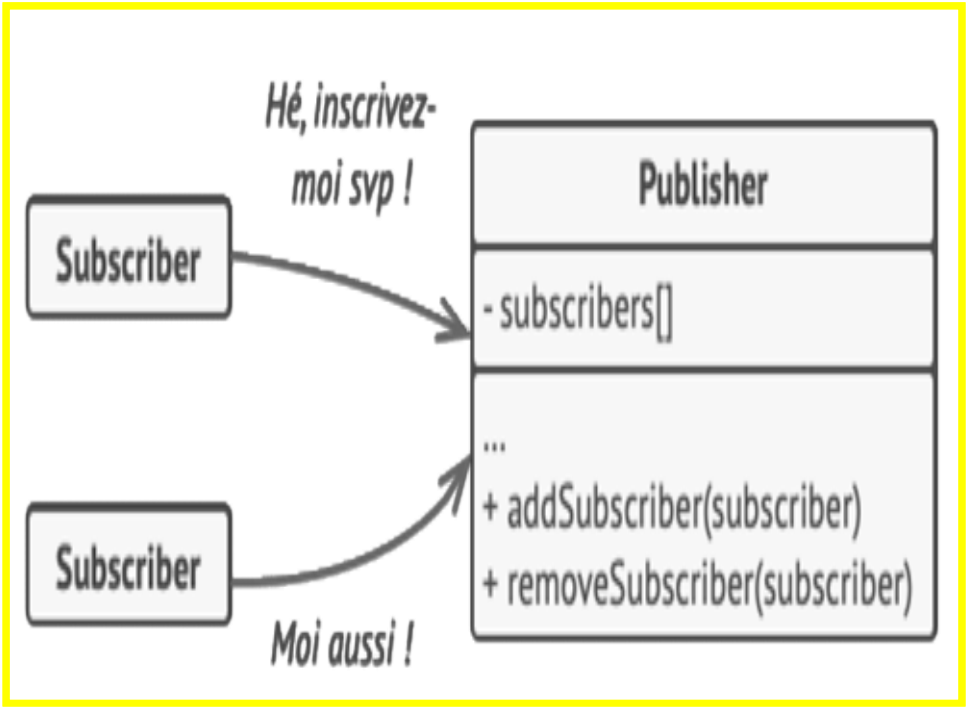
✓Tous les objets qui veulent suivre les modifications apportées au diffuseur sont appelés des ***souscripteurs*** (subscribers).

➢Le patron de conception **Observateur** nous propose d'ajouter un mécanisme de souscription à la classe diffuseur pour permettre aux objets individuels (souscripteurs) de s'inscrire ou se désinscrire de ce diffuseur.

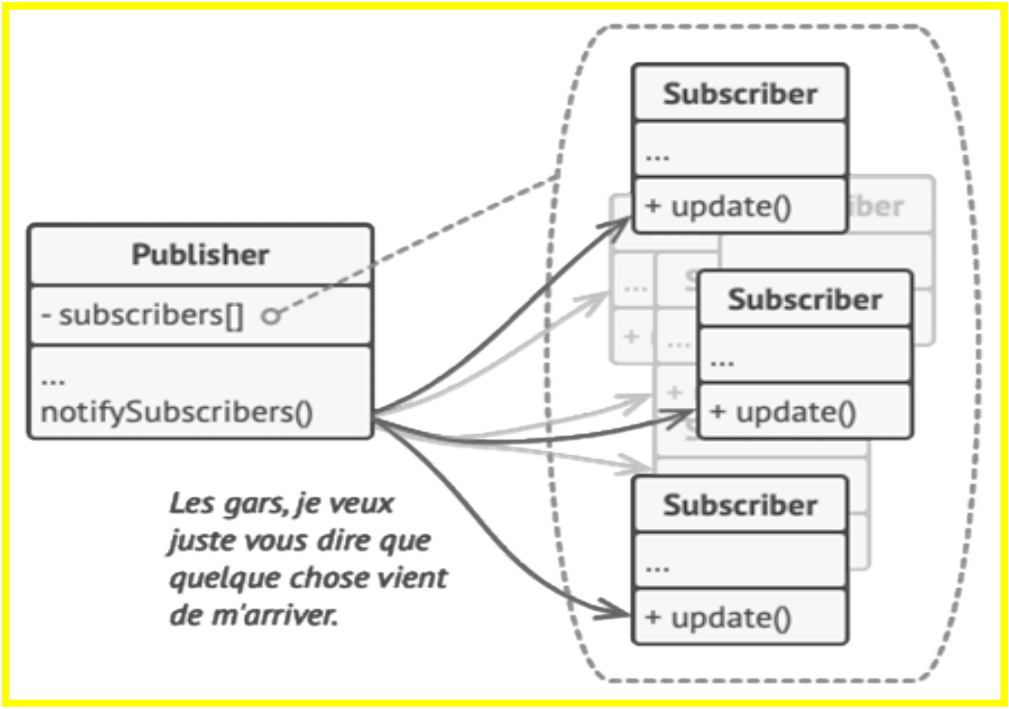
3.Design Patterns Comportementaux

« Observateur »



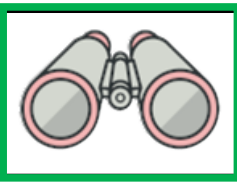


Un mécanisme de souscription qui permet aux individuels de s'inscrire aux notifications des événements

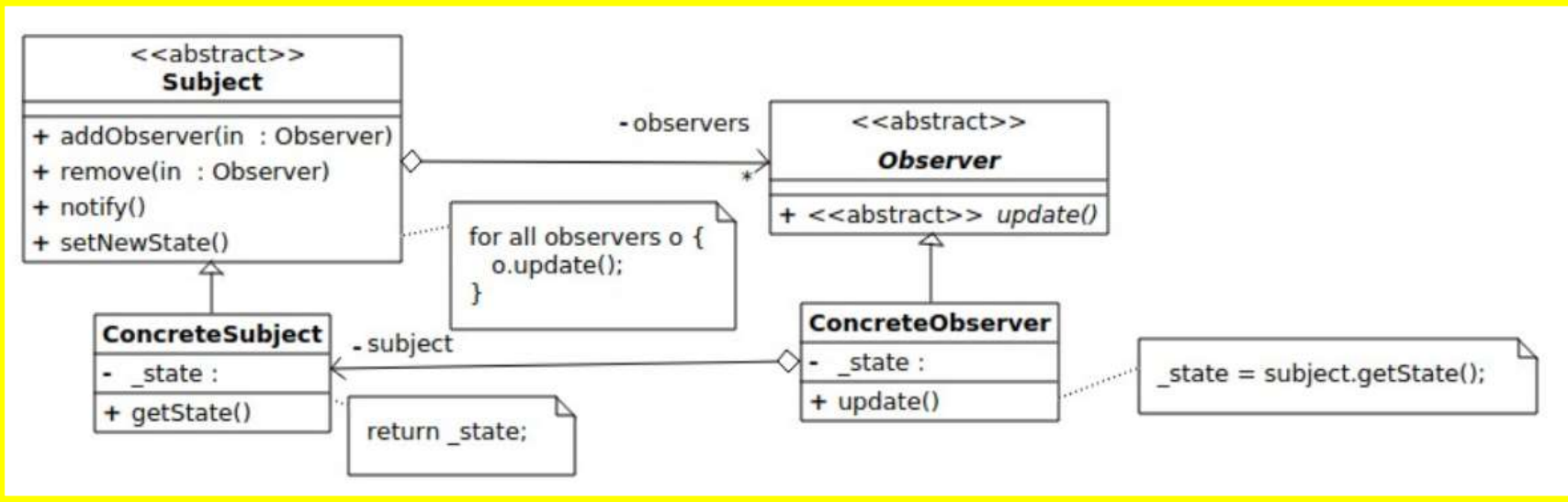


Le diffuseur envoie des notifications aux individuels de s'inscrire aux notifications des événements

3. Design Patterns Comportementaux
« Observateur »



3.4. Structure:



3.4.1 Composants

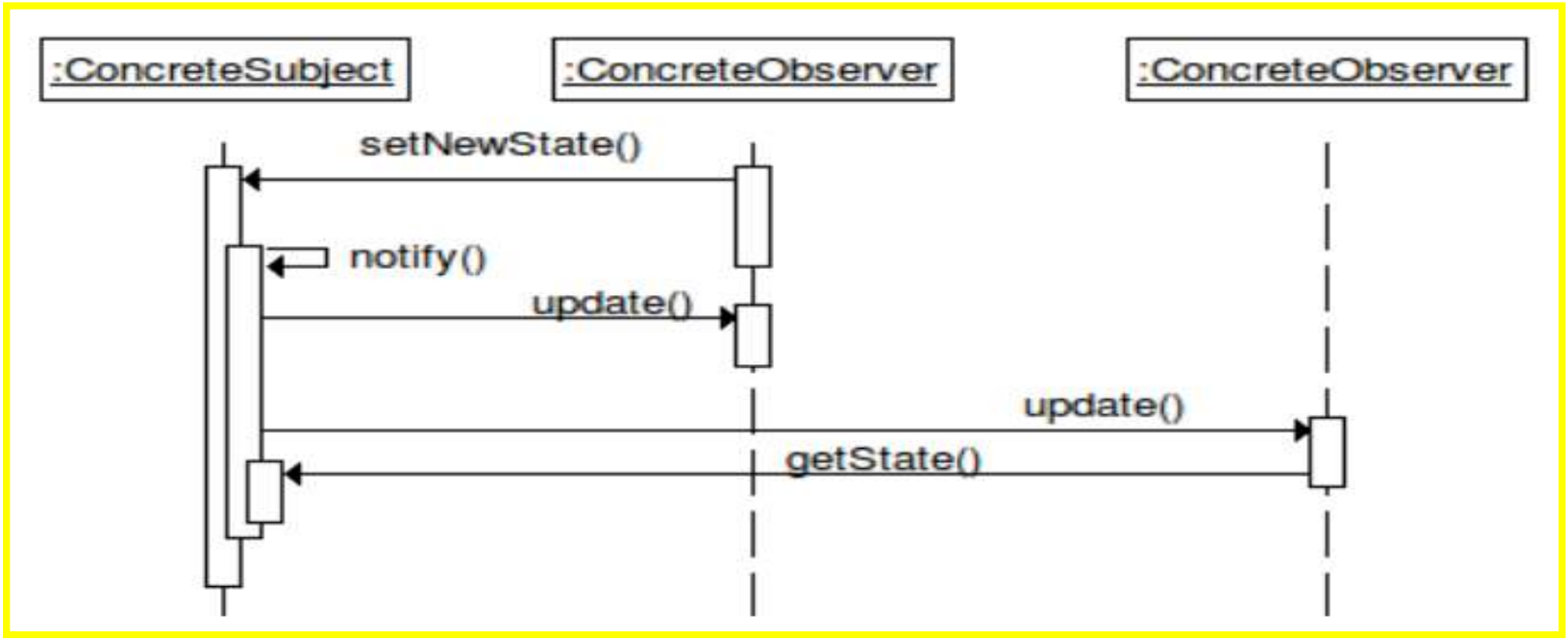
• **Subject** : il connaît les objets observateurs, car ces derniers s'enregistrent auprès de lui. Il doit les avertir dès qu'un événement a eu lieu.
• **ConcretSubject** (eg. magazin) : il mémorise les états qui intéressent les objets observateurs concrets et envoie une notification à ses observateurs par la méthode notify().
• **Observer** : il définit une interface de mise à jour pour les objets qui doivent être notifiés lors d'un changement d'état par la méthode update().

Concrete0bserver (eg. Un Client) **:** Il fait l'implémentation de l'interface de mise à jour de l'observateur pour conserver la cohérence de son état avec le sujet. Il est responsable de son inscription auprès du sujet en utilisant la méthode addObserver() du sujet.

3. Design Patterns Comportementaux

« Obrvateur »

3.5 Collaboration:



Annexe A

La **cohésion** est un principe de conception informatique définissant un degré d'accord entre les différents éléments d'un module

Selon Pressman, il existe sept niveaux de cohésion :

Accidentel : décrivant le niveau le plus faible où le lien entre les différentes méthodes est inexistant ou bien créé sur la base d'un critère futile.

Logique : lorsque les méthodes sont reliées logiquement par un ou plusieurs critères communs.

Temporel : lorsque les méthodes doivent être appelées au cours de la même période de temps.

Procédural : lorsque les méthodes doivent être appelées dans un ordre spécifique.

Communicationnel : lorsque les méthodes manipulent le même ensemble spécifique de données.

Séquentiel : lorsque les méthodes qui manipulent le même ensemble de données doivent être appelées dans un ordre spécifique.

Fonctionnel : réalise le niveau le plus élevé lorsque la classe ou le module est dédié à une seule et unique tâche bien spécifique.

Le niveau accidentel est celui de plus **faible cohésion**, le niveau fonctionnel celui de plus **forte cohésion** ; une bonne architecture logicielle nécessite la plus forte cohésion possible.

Annexe B

- **Le couplage lâche** décrit une technique de couplage dans laquelle deux ou plusieurs composants matériels et logiciels sont attachés ou liés ensemble pour fournir deux services qui ne dépendent pas l'un de l'autre. Ce terme est utilisé pour décrire le degré et l'intention des composants interconnectés mais non dépendants d'un système d'information.

  Le couplage lâche est également appelé couplage faible ou faible.
- **Le couplage lâche** est principalement utilisé dans les réseaux et systèmes d'entreprise pour réduire la quantité et l'intensité des risques rencontrés dans les systèmes hautement dépendants.

Un système avec plusieurs composants est moins susceptible d'avoir des problèmes de performances lorsque ces composants sont couplés de manière **lâche** (Qui est trop peu serré ou tendu, large)

Par exemple, dans une architecture informatique client / serveur, la déconnexion du client du serveur entraînera l'indisponibilité de certaines fonctions, mais le client pourra toujours travailler indépendamment du serveur.

Annexe C

- En programmation orientée objet, Robert C. Martin exprime le **principe de responsabilité unique** comme suit : « une classe ne doit changer que pour une seule raison » (*a class should have only one reason to change*)

Annexe C

Classe abstraite

Une **classe abstraite** est une classe qui **ne peut pas être directement instanciée**. Une classe abstraite **encapsule des attributs et méthodes** qui peuvent être utilisés par les instances des classes qui en héritent. L'intérêt des classes abstraites est de regrouper plusieurs classes sous un même nom de classe (par polymorphisme) ainsi que de décrire partiellement des attributs et méthodes communs à plusieurs classes.

Comme une classe abstraite ne peut pas être instanciée, elle peut encapsuler des méthodes dites abstraites, c'est-à-dire dont le traitement n'est pas défini dans la classe. Par contre, les classes héritant d'une classe abstraite doivent obligatoirement redéfinir ses méthodes abstraites ou alors être elles-mêmes abstraites.

Annexe C

Classe abstraite

En UML, le mot-clé {abstrait} (ou {abstract}) est accolé aux classes et méthodes abstraites. Une autre manière souvent usitée de représenter ces méthodes ou classes est d'écrire leur nom en italique.

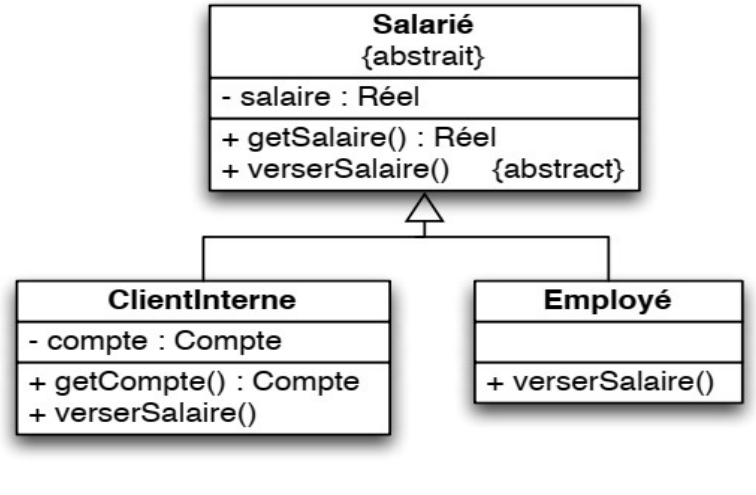


FIGURE 1 – Exemple de classe abstraite

```
public abstract class Salarie {
  private double salaire;

  public double getSalaire() {
    return salaire;
  }

  public abstract void verserSalaire();
}
```

Annexe D

Classe Interface : Une **interface** décrit des objets mais uniquement en terme de méthodes abstraites. Une interface **ne peut contenir** **ni attribut**, **ni méthode implémentée**. Le terme d'héritage n'est pas utilisé entre classes et interfaces : on dit qu'une classe **implémente** une interface. Lorsqu'une classe implémente une interface, elle doit redéfinir toutes ses méthodes abstraites ou bien être abstraite.
Une interface peut hériter d'une autre interface.

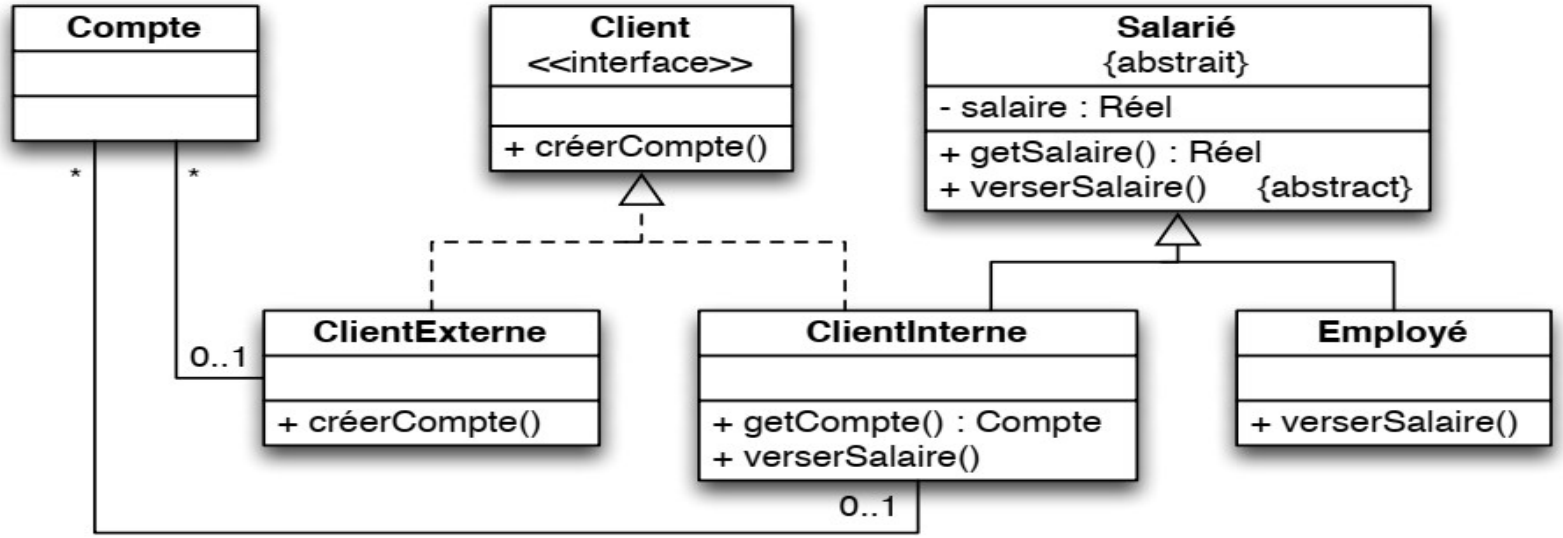


FIGURE 2 – Exemple d'interface

Annexe D

Classe Interface :L'intérêt d'une interface est de regrouper plusieurs classes par polymorphisme. De plus, une classe peut implémenter plusieurs interfaces différentes.
En UML, une interface est décrite par le mot-clé **«interface»** et la relation d'implémentation entre une classe et une interface est représentée par une flèche d'héritage en pointillé.

En Java, le code source de l'interface Client se situe dans un fichier Client.java ayant pour contenu :

```
public interface Client {

  public void creerCompte();

}
```

Un exemple d'héritage de la classe abstraite Salarie et d'implémentation de l'interface Client:

```
public class ClientInterne extends Salarie implements Client {
  private Compte compte;

  public void creerCompte() {
    compte = new Compte();
  }

  public void verserSalaire() {
    compte.credite(salaire);
  }
}
```

Bibliographie

1. Chapitre 3 : Catalogue de patrons de conception ‘’Régis Clouard‘’
2. https://refactoring.guru/fr/design-patterns.