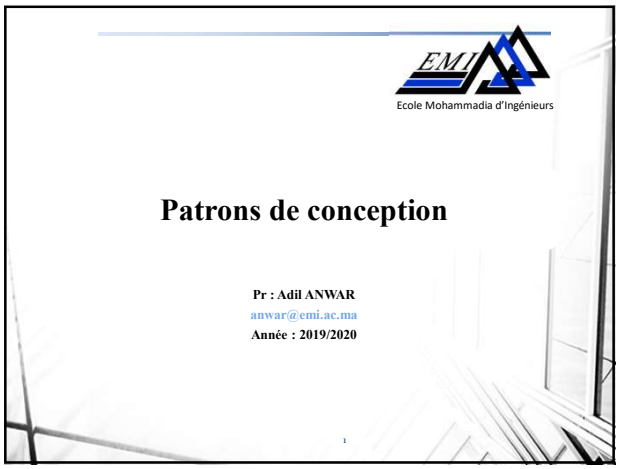



# Plan

- Rappel des principes de base de la conception objet
- Principes S.O.L.I.D : écrire du bon code
  - Principe de responsabilité unique
  - Principe d'ouverture/fermeture
  - Principe de substitution de Liskov
  - Principe de séparation d'interfaces
  - Principe d'inversion de dépendances
- Design Patterns (GOF)

## Principes de la conception orientée objet

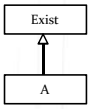
- les **principes de la conception orientée objet** (OOD) vous permettent de développer des applications bien conçues , maintenables, réutilisables et évolutives.
  - **Programmer pour des interfaces** : Limiter le couplage et protéger des variations en faisant abstraction de l'implémentation des objets
  - **Composer au lieu d'hériter** : Limiter le couplage en utilisant la composition (boite noire) au lieu de l'héritage (boite blanche) pour déléguer une tâche à un objet.
  - **Protection des variations** : Identifier les points de variation et d'évolution, et séparer ces aspects de ceux qui demeurent constants.
  - **Indirection** : Limiter le couplage et protéger des variations en ajoutant des objets intermédiaires (ex classes Façade)

## Retour sur l'héritage

- Quand utiliser l'héritage?
  1. Factorisation du code : Lorsque deux ou plusieurs classes partagent plusieurs attributs (et méthodes), elles font les mêmes choses
  2. Réutilisation : une classe existante fait quasiment ce que vous voulez faire
  3. Imposer un cadre : vos classes proposent un noyau qui doit être complété

## Retour sur l'héritage

- Quand utiliser l'héritage?
  - Réutilisation : une classe existante fait quasiment ce que vous voulez faire
  - Avant de coder une nouvelle classe, on peut chercher une classe existante
  - Hériter de cette classe
    - Ajouter méthodes et attributs
    - Exploiter le polymorphisme




## Retour sur l'héritage

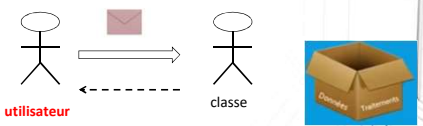
- Quand utiliser l'héritage?
  - Imposer un cadre:
  - Vous voulez fournir du code dont l'unique objectif est d'être réutilisé mais aussi d'être complété.
  - Vous faite une classe qui contient le code qui va être réutilisé dont l'objectif qu'elle soit hérité

## Retour sur les interfaces

- la classe précise
  - Le type des objets (données, traitements)
  - Le code des traitements
  - La façon dont les objets sont créés (instance)

utilisateur

développeur

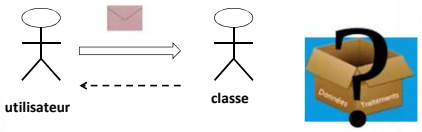






## Retour sur les interfaces

- L'utilisateur n'a besoin de savoir que le type des traitements (**vue utilisateur**)
  - Quels traitements ?
  - Quels paramètres ?







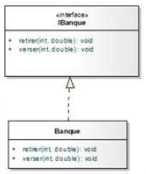
## Retour sur les interfaces

- Définition d'un ensemble de traitements (typage)
  - Doit être réalisée (implémentée) par des classes non abstraites
  - Peut hériter d'une autre interface
- Pourquoi des interfaces ?
  - On utilise des interfaces pour établir des types abstraits (Utilisation similaire aux classes abstraites)
  - En Java : une classe ne peut hériter de plus d'une classe, mais elle peut réaliser plusieurs interfaces
  - Sert à l'implémentation d'autres classes et non à leur structure


```
public interface IBanque {
        void retirer(int numero,double montant)
        void verser(int numero, double montant)
```

## Retour sur les interfaces

- L'utilisateur ne connait que l'interface
- Le typage des objets est donc celui de l'interface exploité par l'utilisateur
- L'interface n'est que la vue utilisateur
- Derrière chaque interface se cache une classe
- La classe réalise l'interface (implements en java)




```
public class Banque implements IBanque{

    @Override
    public void verser(int numero, double montant)
            // ajouter votre code ici
        }
```

## Principes de la conception orientée objet

Appliquer les principes de forte cohésion et de faible couplage

**Cohésion et faible couplage**

**Cohésion**

- Les classes et les modules qui ont le même focuse (**cohésion**) et qui ne sont pas très dépendants (ou couplés) avec d'autres classes ou modules sont généralement **faciles** à utiliser, **réutilisables** et **maintenables**
- La cohésion se réfère à la **responsabilité** d'une classe ou d'un module.
- Une forte cohésion se réfère à une classe ou à un module ayant un focus et une responsabilité bien définie.
- La faible cohésion se réfère à une classe ou un module qui n'a pas une responsabilité bien définie

## Principes de la conception orientée objet

Appliquer les principes de forte cohésion et de faible couplage

**Cohésion et faible couplage**

**Cohésion**

- Un éditeur de livre est supposé éditer le contenu du livre, gérer le processus d'édition de livres et s'approcher à des nouveaux auteurs.
- **Faible Cohésion**

```
class Editor{
    public void editBooks() {}
    public void manageBookPrinting() {}
    public void reachOutToNewAuthors() {}
}
```

**Low cohesion; Editor is performing diverse set of unrelated tasks**

- **Forte Cohésion**

```
class Editor{
    public void useEditTools() {}
    public void editFirstDraft() {}
    public void clearEditingDoubts() {}
}
```

**High cohesion; Editor is performing multiple but related tasks.**

## Principes de la conception orientée objet

Appliquer les principes de forte cohésion et de faible couplage

**Cohésion et faible couplage**

**Couplage**

- Le couplage se réfère au niveau de dépendance entre une classe ou un module avec d'autres classes ou modules.
- Si une classe B interagit avec une autre classe A en utilisant son interface (méthodes publiques), les classes B et A sont faiblement couplées (**loosely coupled**)
- Si la classe B peut accéder et manipuler la classe A en utilisant ses membres non publics, ces classes sont fortement couplées (**tightly coupled**).

## Principes de la conception orientée objet

Appliquer les principes de cohésion, de faible couplage

**Cohésion et faible couplage**

**Couplage**

```
class Author {
    String name;
    String skypeID;
    public String getSkypeID() {
        return skypeID;
    }
}

class Editor{
    public void clearEditingDoubts(Author author) {
        setUpCall(author.skypeID);
        converse(author);
    }
    void setUpCall(String skypeID) { /* */}
    void converse(Author author) {/* */}
}
```

Tight coupling; nonpublic variable skypeID is referred to outside its class Author.

## Principes de la conception orientée objet

Appliquer les principes de forte cohésion et de faible couplage

**Cohésion et faible couplage**

**Couplage**

- Que se passe-t-il, par exemple, si un programmeur modifie le nom de la variable skypeID par skypeName?
- Le code de la classe Editor ne se compilera pas.
- Tant que l'interface publique d'une classe reste la même, le détail d'implémentation pourra être modifié sans impacte.

## Principes de la conception orientée objet

Appliquer les principes de forte cohésion et de faible couplage

**Cohésion et faible couplage**

**Couplage**

```
class Author {
    String name;
    String skypeName;
    public String getSkypeID() {
        return skypeName;
    }
}
class Editor{
    public void clearEditingDoubts(Author author) {
        setUpCall(author.getSkypeID());
        converse(author);
    }
    void setUpCall(String skypeID) { /* */}
    void converse(Author author) {/* */}
}
```

Change in instance variable name won't affect classes that access this method

Loose coupling; public method getSkypeID() accesses Author's skypeName.

## Principes de la conception orientée objet

Appliquer les principes de cohésion, de faible couplage

**Cohésion et faible couplage**

**Couplage**

- Les interfaces favorisent également le couplage faible entre les classes et les modules

```
interface Author {
    String getSkypeID();
}
class TechnicalAuthor implements Author{
    String name;
    String skypeName;
    public String getSkypeID() {
        return skypeName;
    }
}
class Editor{
    public void clearEditingDoubts(Author author) {
        setUpCall(author.getSkypeID());
        converse(author);
    }
    void setUpCall(String skypeID) { /* */}
    void converse(Author author) {/* */}
}
```

1. interface Author
2. Class TechnicalAuthor implements Author.
3. Loose coupling; method clearEditingDoubts() uses interface to access concrete implementations

## Principes de la conception orientée objet

Appliquer les principes de la composition d'objets

- Dans la programmation orientée objet, l'héritage et la composition sont deux moyens qui permettent la réutilisation des classes
- Comment utiliser les fonctionnalités existantes d'une classe?
- L'héritage est-t-il le meilleure moyen ?
- La composition d'objet vous permet d'utiliser les fonctionnalités existantes des classes sans les étendre. L'approche est simple: créer et utiliser des objets d'autres classes dans votre propre classe
- **Utilisez la composition pour assembler des comportements d'autres classes**
  - La composition vous permet d'utiliser le comportement d'une famille d'autres classes et de **changer** ce comportement **pendant** le fonctionnement de l'application

## Principes de la conception orientée objet

la composition d'objets vs l'héritage

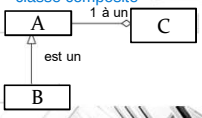
- Exemple

```
public class A {
    private int v1 =2;
    private int v2=3;

    public int opA() {
        return v1+v2;
    }
}
```

```
public class B extends A {
    private int v3= 5;

    public void opB() {
        System.out.println(super.opA()*v3);
    }
}
```

```
public class C {
    private int v3=5;
    private A a= new A();
    public void opC() {
        System.out.println(a.opA()*v3);
    }
}
```

A = classe partie
C = classe composite

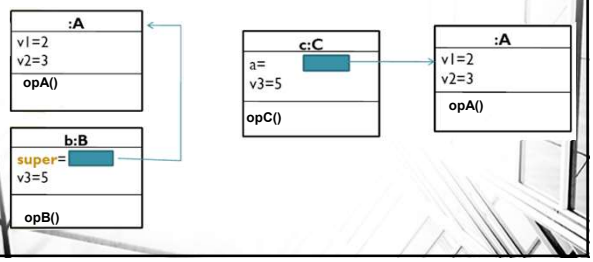



## Principes de la conception orientée objet

la composition d'objets vs l'héritage

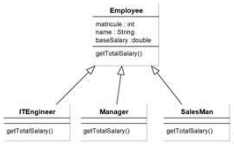
- B b=**new B();**
- b.meth2();

C c=**new C();**
c.meth2();




## Héritage vs Composition

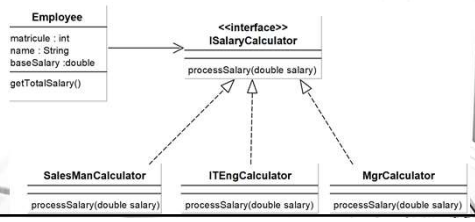
- Exemple : Calcul du salaire d'un employé
  - Calculer le salaire d'un employé en fonction de sa fonction (commercial, ingénieur IT, manager, etc. )
  - Règles métier :
    - les managers perçoivent une commission de responsabilité fixe de 20%
    - Les ingénieurs IT perçoivent une prime par projet de 5%
    - Les commerciaux perçoivent une commission par opération de vente de 2%
  - Solution 1 : utiliser l'héritage




  - Et si la règle de calcul du salaire change fans le futur ?
  - ➔ Principe de protection des variations

## Héritage vs Composition

- Solution 2 : la composition
- **Principe 1** : **la composition,** elle permet de créer un objet employé par instanciation de la classe employé et association (composition) de propriétés qui lui correspondent(ex. la classe de calcul d'un salaire pour un ingénieur IT).
- **Principe 2 : Protection des variations** : Identifier les points de variation et d'évolution, et séparer ces aspects de ceux qui demeurent constants.




## Code source de l'exemple

- L'interface *ISalaryCalculator*

```
public interface IEmail {
     void sendEmail(String content);
}
```

- La classe *ManagerSalaryCalcutor* qui implémente le comportent de calcul du salaire d'un manager

```
public class ManagerSalaryCalculator implements ISalaryCalculator {
    public double salary(double salary) {
            return salary*1.2;
    }
}
```

- La classe *CommercialSalaryCalculator* qui implémente le comportement de calcul du salaire pour un commercial

```
public class CommercialSalaryCalculator implements ISalaryCalculator {
    public double salary(double salary) {
        return salary*1.1;
    }
}
```

## Code source de l'exemple

- La classe employé qui utilise le comportement de calcul du salaire par délégation aux classes implémentant l'interface ISalaryCalculator

```
public class Employe {
    private double salaire;
    private ISalaryCalculator calculator;
    public Employe(double salaire) {
        this.salaire = salaire;
    }
    public double getSalaire() {
        return this.salaire;
    }

    public void setCalculaor(ISalaryCalculator calc)
        this.calculator = calc;
    }
    public double calculSalaire() {
        return calculator.salary(this.salaire);
    }
}
```

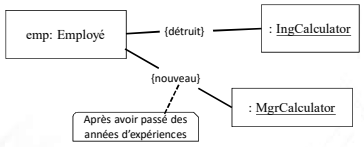
## Code source de l'exemple

- Classe Main pour le test

```
public class TestSalaire {

    public static void main(String[] args) {
        Employe emp = new Employe(5000);

        ISalaryCalculator managerCalculator = new ManagerSalaryCalculator();
        ISalaryCalculator commercialCalculator = new CommercialSalaryCalculator();

        //affectation du comportement de calcul de salaire "manager"
        emp.setCalculaor(managerCalculator);
        System.out.println(emp.calculSalaire());

        //affectation du comportement de calcul de salaire "commercial"
        emp.setCalculaor(commercialCalculator);
        System.out.println(emp.calculSalaire());
    }
}
```

## Héritage vs composition: Comparaison

- La généralisation suppose la création préalable d'une sous-classe de Employé
- **La modification d'une caractéristique**
  - dans le cadre d'une généralisation, la modification de la fonction par exemple implique la création d'un nouvel objet instance d'une sous-classe employé, la définition de toutes ses caractéristiques (même celles qui ne sont pas liées à la modification) et la destruction de l'ancien objet.
  - Dans le cadre de la composition, les objets de type Employe peuvent changer leurs propriétés d'une façon dynamique.




## Héritage vs Composition : Comparaison

2. **L'ajout d'une nouvelle caractéristique**
   - Dans le cas de la composition, l'extension d'une caractéristique existante (par le statut de l'employé (salarié de l'entreprise, contractuel) ) se fait en créant une nouvelle sous-classe, qui n'induit aucune modification au niveau des instances d'employés déjà définies.
   - Dans le cas de la composition, l'ajout d'une nouvelle caractéristique (par exemple, le statut de l'employé (salarié de l'entreprise, contractuel) .) est réalisé par l'association d'une classe abstraite **Statut Employé** à la classe **employé**.
   - Dans le cas de l'héritage, l'ajout d'une nouvelle caractéristique conduit à la modification des sous-classes terminales.

## Héritage vs Composition

- Pour résumé : la composition apporte davantage de flexibilité et de réutilisation que l'héritage.
  - L'héritage est une structure rigide mais la propagation des attributs et des opérations vers les sous-classes est automatique
  - l'héritage offre un mécanisme simple et puissant qui permet de modifier et d'étende le code existant.
  - La composition d'objets est une construction plus souple mais la propagation des propriétés doit être réalisée manuellement.
  - Elle permet également la mise ne œuvre de la généralisation multiple avec les langages qui ne possèdent que l'héritage simple ex Java.
- Attention toutefois à ne pas abuser de la composition :
  - La flexibilité et l'intelligibilité sont en opposition : plus la structure est flexible et plus elle est complexe à comprendre; plus elle est statique et plus elle est intelligible. Ainsi, il ne faut pas introduire de flexibilité inutile.

Ecole Mohammadia d'Ingénieurs

# Patrons de conception

Pr : Adil ANWAR
anwar@emi.ac.ma
Année : 2019/2020

1

## Plan

- Principes de base de la conception objet : GRASP
- Principes S.O.L.I.D : écrire du bon code
  - Principe de responsabilité unique
  - Principe d'ouverture/fermeture
  - Principe de substitution de Liskov
  - Principe de séparation d'interfaces
  - Principe d'inversion de dépendances
- Design Patterns (GOF)

## Les principes S.O.L.I.D

- écrire du bon code !
  - **S**ingle responsibility principle (SRP)
  - **O**pen-closed principle (OCP)
  - **L**iskov substitution principle (LSP)
  - **I**nterface segregation principle (ISP)
  - **D**ependency inversion principle (DIP)

## Principe de la responsabilité unique (SRP)

- Chaque classe dans votre application ne doit avoir qu'une seule raison de changer (Robert Martin).
  - Une responsabilité = une raison de changer
  - Séparer les responsabilités couplées en classes distinctes
  - Chaque classe ou module doit se concentrer sur une seule tâche à la fois.
  - tout ce qui est définit sur cette classe devrait être lié à cette unique tâche
  - En appliquant le principe SRP, les classes deviennent plus petites et le code est plus propre.

[1]. Robert C. Martin.

## Principe de la responsabilité unique (SRP)

- Considérons l'exemple simple d'une interface IUser qui déclare 4 méthodes utilisés dans le processus d'inscription et d'authentification des utilisateurs :
  - login : une méthode qui permet à un utilisateur de se connecter avec un nom d'utilisateur et un mot de passe.
  - register : une méthode qui créer un nouveau utilisateur.
  - logError : méthode de journalisation des erreurs qui affiche dans la console les erreurs de l'application.
  - sendEmail: méthode qui implémente la fonctionnalité d'envoi de mails aux utilisateurs

```
public interface IUser {
    boolean login(String userName, String password);
    boolean register(String userName, String password, String userEmail);
    void logError(String error);
    void sendEmail(String content);
}
```

## Principe de la responsabilité unique (SRP)

- Analysons l'interface IUser :
  - Nous avons regroupé dans cette interface 4 fonctionnalités plus ou moins indépendantes
  - Supposons qu'il y'a un changement dans l'exigence de journalisation des erreurs, et on veut ajouter la journalisation dans un fichier d'érreurs au lieu de l'afficher dans la console→ changement dans la classe et recompilation des classes dépendantes mêmes celles qui ne sont pas concernées par la journalisation
  - Supposons qu'il y'a un changement dans la logique d'envoi de mail, par exemple utiliser une API basée sur https ou lieu du protocole SMTP.
  - À chaque fois on sera obligé de recompiler toute la classe et celles qui ont dépendent → regroupement de plusieurs responsabilités distinctes dans une seule classe - une violation de SRP

6

## Principe de la responsabilité unique (SRP)

- Solution :
  - Séparer l'interface IUser en plusieurs interfaces qui respectent le principe de la responsabilité unique

1. Responsabilité d'inscription et d'authentification

```
public interface IUser {
   boolean login(String userName, String password);
   boolean register(String userName, String password, String userEmail);
}
```

2. Responsabilité de journalisation

```
public interface ILogger {
     void logError(String error);
}
```

3. Responsabilité d'envoie de mail

```
public interface IEmail {
    void sendEmail(String content);
}
```

7

## Principe de la responsabilité unique (SRP)

- Exemple
  - Nous avons développé une application de gestion des employés et nous devons trier les employés par nom, age ou salaire.
  - Solution :
  - On peut donc créer une classe employee qui implémente l'interface Comparator.

```
class Employe implements Comparator {
   int compare(Object o) { ... } }
```

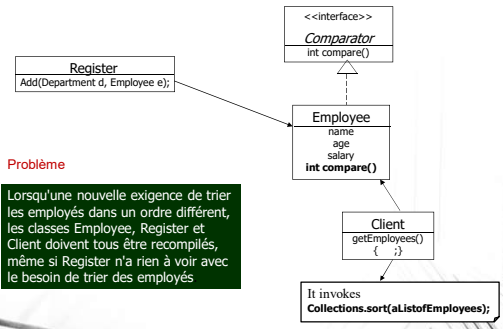
8

## Principe de la responsabilité unique (SRP)

- Discussions:
  - Employé est une entité métier, on ne connait pas dans quel ordre elle doit être triée puisque l'ordre de tri est imposé par une classe client.
  - Pire: chaque fois que les employés doivent être triés différemment, nous devons recompiler la classe Employé et tous ses classes clients.
  - Cause des problèmes: nous avons regroupés deux responsabilités distinctes (c.-à-d. L'employé en tant qu'entité métier avec la fonctionnalité de trie) dans une seule classe - une violation de SRP

9

## Principe de la responsabilité unique (SRP)




Problème

Lorsqu'une nouvelle exigence de trier les employés dans un ordre différent, les classes Employee, Register et Client doivent tous être recompilés, même si Register n'a rien à voir avec le besoin de trier des employés

## Principe d'ouverture / fermeture (OCP)

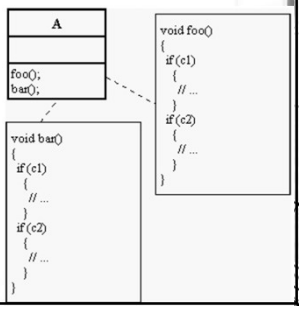
- Vous devriez être capable d'étendre le comportement d'une classe sans le modifier(Robert C. Martin)
- Les entités logicielles doivent être **ouvertes à l'extension**
  - le code est extensible pour proposer des comportements qui n'étaient pas prévus lors de sa création.
- mais **fermées aux modifications**
  - Le code a été écrit et testé, on n'y touche pas pour éviter les régressions.
  - Les extensions sont introduites sans modifier le code existant
- Pour rendre une classe ouverte pour l'extension, fermée pour la modification, coder avec des **interfaces** (ou des **classes abstraites**), plutôt qu'avec des implémentations (classes concrètes).

## Principe d'ouverture / fermeture (OCP)

- **L'abstraction comme moyen d'ouverture/fermeture**
- L'ouverture/fermeture se pratique en faisant reposer le code « fixe » sur une abstraction du code amené à évoluer.
- En d'autres termes, l'OCP consiste à **séparer le stable du variable**,
- ➔mais il faut savoir identifier ce qui sera stable et variable pour pouvoir les séparer.
- Parmi les mécanismes principaux qui permettent de mettre en place l'abstraction préconisée par l'OCP.
  - L'utilisation des classes d'interface (classes abstraites en Java, C++, interfaces en Java )

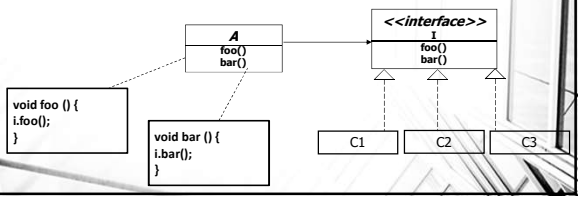
## Principe d'ouverture / fermeture (OCP)

- Exemple d'introduction
  - Utilisation de la « délégation abstraite »
    - A gère les classes c1 et c2.
    - Si un nouveau cas c3 doit être géré, il faut modifier le code de A en conséquence (opérations A.foo() et A.bar()) :



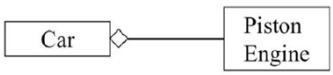

## Principe d'ouverture / fermeture (OCP)

- Exemple d'introduction
  - Le code de **A** peut être ouvert aux extensions et fermé aux modifications en introduisant une interface I dont dérivent des classes **C1** et **C2** et aussi C3:
  - Puisque A repose uniquement sur l'interface I, il devient possible d'ajouter un nouveau cas c3 sous la forme d'une classe C3 dérivée de I, sans avoir à modifier A.



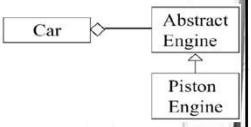

## Principe d'ouverture / fermeture (OCP)

- Exemple :
- Comment faire en sorte que la voiture aille plus vite à l'aide d'un turbo?
  - Il faut changer la voiture avec la conception actuelle…




```
public class Car {
    private PistonEngine engine;

public Car(PistonEngine engine) {
    this.engine = engine;
}
```

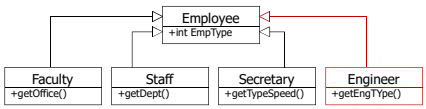
## Principe d'ouverture / fermeture (OCP)




```
public class Car {
    private AbstractEngine engine;
public Car(AbstractEngine engine) {
    this.engine = engine;
}

public class PistonEngine extends AbstractEngine{

}
```

- Une classe ne doit pas dépendre d'une classe Concrète
  - Elle doit dépendre d'une classe abstraite
  - ...et utiliser le polymorphisme

## Principe d'ouverture / fermeture (OCP)

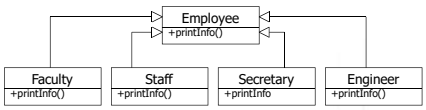



**Un exemple de Ce qu'il ne faut pas faire! Qu'est-ce qui ne va pas avec ce code?**

```
void printEmpRoster(Employee[] emps) {
    for (int i; i<emps.size(); i++) {
        if (emps[i].empType == FACULTY)
            printfFaculty((Faculty)emps[i]);
        else if (emps[i].empType ==STAFF)
            printStaff((Staff)emps[i]);
        else if (emps[i].empType == SECRETARY)
            printSecretary((Secretary)emps[i]);
    }
}
```

**Et si nous devons ajouter un ingénieur ??**

## Principe d'ouverture / fermeture (OCP)




```
void printEmpRoster(Employee[] emps) {
    for (int i; i<emps.size(); i++) {
        emps[i].printInfo();
    }
}
```

**Lorsque la classe Engineer est ajoutée, la méthode printEmpRoster () n'a même pas besoin de recompiler. PrintEmpRoster () est ouverte à l'extension, fermée à la modification.**

## Principe d'ouverture/fermeture(OCP) : allons plus loin (1)

- Le but de cette méthode est de calculer le prix total d'un ensemble de produits «Product»

```
public double totalPrice(Product[] products) {
        double total = 0.0;
        for (int i=0; i< products.length; i++) {
                total += products[i].getPrice();
        }
        return total;
}
```

## Principe d'ouverture/fermeture(OCP) : allons plus loin (2)

- «Mais le département comptabilité décrète que les pièces de la carte mère et les pièces de mémoire devraient avoir une prime appliquée lors de la détermination du prix total.»
- Que pensez-vous du code suivant?

```
public double totalPrice(Product[] products) {
        double total = 0.0;
        for (int i=0; i< products.length; i++) {
                if (products[i] instanceof Motherboard)
                        total += (1.45 * products[i].getPrice());
                else if (products[i] instanceof Memory)
                        total += (1.27 * products[i].getPrice());
                else
                        total += products[i].getPrice();
        }
return total;
}
```

## Principe d'ouverture/fermeture(OCP) : allons plus loin (3)

- Voici des exemples de classes Product et ConcreteProduct
- // Class Product is the superclass for all parts.

```
public class Product {
protected double price;
public Product(double price) (this.price = price;}
public void setPrice(double price) {this.price = price;}
public double getPrice() {return price;}
}
// Class ConcreteProduct implements a product for sale.
// Pricing policy explicit here!
public class ConcreteProduct extends Product {
public double getPrice() {
// return (1.45 * price); //Premium
return (0.90 * price); //Labor Day Sale
}
```

But now we must modify each subclass of Part whenever the pricing policy changes!

## Principe d'ouverture/fermeture(OCP) : allons plus loin(4)

- Une meilleure idée est d'avoir une classe PricePolicy qui peut être utilisée pour fournir différentes stratégies de tarification :
  - Principe de conception :

  ➔Protection des variations : Identifier les points de variation et d'évolution, et séparer ces aspects de ceux qui demeurent constants.

```
// The Product class now has a contained PricePolicy object.
    public class Product {
    private double price;
    private PricePolicy pricePolicy;
    public void setPricePolicy(PricePolicy pricePolicy) {
          this.pricePolicy = pricePolicy;}
    public void setPrice(double price) {this.price = price;}
    public double getPrice() {return pricePolicy.getPrice(price);}
    }
```

## Principe d'ouverture/fermeture(OCP) : allons plus loin (5)

```
/**
* Class PricePolicy implements a given price policy.
*/
    public class PricePolicy {
    private double factor;
    public PricePolicy (double factor) {
      this.factor = factor;
    }
    public double getPrice(double price) {return price * factor;}
    }
```

- D'autres politiques comme un calcul de la ristourne par «seuils» est maintenant possible ...
- Avec cette solution, nous pouvons définir **dynamiquement** des stratégies de tarification au moment de l'exécution en modifiant l'objet *PricePolicy* auquel un objet Product existant fait référence.
- Bien sûr, dans une application réelle, le prix d'un produit et sa politique de prix associée peuvent être contenus dans une base de données.

## Principe d'ouverture/fermeture(OCP): conclusion

- Identifier les points d'ouverture/fermeture en fonction des
  - Besoins d'évolutivité exprimés par le client
  - Besoins de flexibilité pressentis par les développeurs
  - Changements répétés constatés au cours du développement
- Il n'est pas possible que tous les modules d'un système logiciel satisfirent l'OCP, mais nous devons essayer de minimiser le nombre de modules qui ne le satisfont pas.
- Le principe ouvert-fermé est vraiment le cœur de la conception OO.
- La conformité à ce principe donne le plus haut niveau de réutilisabilité et de maintenabilité.

## Principe de séparation des interfaces (ISP)

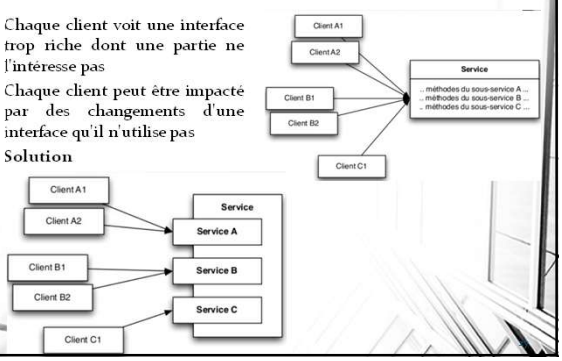
- De nombreuses interfaces spécifiques au client valent mieux qu'une interface à usage général. (Robert C. Martin).
- «Les clients ne devraient pas être obligés de dépendre de méthodes qu'ils n'utilisent pas.» - R. Martin
- Lorsque nous regroupons des fonctions pour différents clients en une seule interface / classe, nous créons un couplage inutile entre les clients.
- Lorsqu'une classe client provoque le changement d'interface, tous les autres classes clients sont obligés de se recompiler.
- Un client doit avoir des interfaces avec uniquement ce dont il a besoin
- Incite à avoir des interfaces petites pour ne pas forcer des classes à implémenter les méthodes qu'elles ne veulent pas.

## Principe de séparation des interfaces (ISP)

- Les clients ne doivent pas être forcés de dépendre d'interfaces qu'ils n'utilisent pas
- Chaque client ne doit «voir» que les services qu'il utilise réellement
- Evite une tentation courante : accumuler dans une classe un ensemble de services sous prétexte que la classe contient les informations nécessaires
  - Solution : utiliser des interfaces différentes par type de client différent

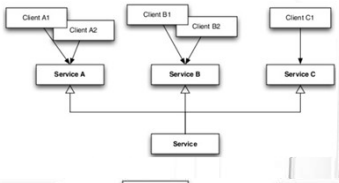
## Principe de séparation des interfaces (ISP)

- Chaque client voit une interface trop riche dont une partie ne l'intéresse pas
- Chaque client peut être impacté par des changements d'une interface qu'il n'utilise pas
- Solution

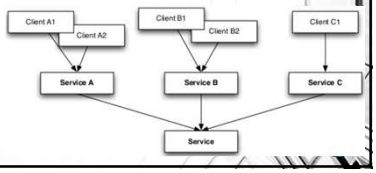


## Principe de séparation des interfaces (ISP)
### Mise en œuvre

- Par héritage multiple (qd permis)



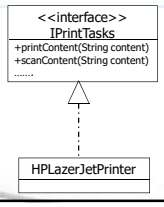

- Par classe d'adaptation




## Principe de séparation des interfaces (ISP)

Exemple d'illustration

```
public interface IPrintTasks {
    boolean printcontent(String content);
    boolean scanContent(String content);
    boolean faxContent(String content);
    boolean photocopyContent(String content);
}
```







## Principe de séparation des interfaces (ISP)

- Exemple d'illustration

```
public class HPLazerJetPrinter implements IPrintTasks {
    public boolean printcontent(String content) {
        System.out.println("print done !");
        return true;
    }
    public boolean scanContent(String content) {
        System.out.println("scan done !");
        return true;
    }
    public boolean faxContent(String content) {
        System.out.println("fax done !");
        return true;
    }
    public boolean photocopyContent(String content) {
        System.out.println("photocopy done !");
        return true;
    }
```

## Principe de séparation des interfaces (ISP)

Exemple d'illustration

- Problème 1 : Supposons que nous avons une deuxième classe *EPsonPrinter* qui souhaite implémenter l'interface IPrintTasks mails elle ne dispose pas des fonctionnalités de fax ou de photocopie.

```
public class EPsonPrinter implements IPrintTasks {
    public boolean printcontent(String content) {
        System.out.println("print done !");
        return true;
    }
    public boolean scanContent(String content) {
        System.out.println("scan done !");
        return true;
    }
    public boolean faxContent(String content)  {
        // throw new Exception("Unsupported function");
        return false;
    }
    public boolean photocopyContent(String content) {
        // throw new Exception("Unsupported function");
        return false;
    }
}
```

## Principe de séparation des interfaces (ISP)

Exemple d'illustration

- Problème 2 : Supposons que nous souhaitons ajouter une méthode *printDuplexContent* dans l'interface pour supporter une nouvelle fonctionnalité d'impression en mode recto verso.

```
public interface IPrintTasks {
    boolean printcontent(String content);
    boolean scanContent(String content);
    boolean faxContent(String content);
    boolean photocopyContent(String content);
    boolean printDuplexContent(String content);
}
```

## Principe de séparation des interfaces (ISP)

Exemple d'illustration

- Pour résoudre ce problème, il suffit d'appliquer le principe ISP qui consiste à faire un refactoring de l'interface IPrintTasks en trois interfaces plus petites :
- Une interface pour les fonctionnalités de base

```
public interface IPrintBasicTasks {
    boolean printcontent(String content);
    boolean scanContent(String content);
    boolean photocopyContent(String content);
}
```

- Une interface pour la fonction se scan

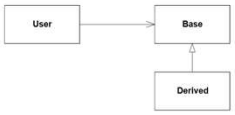
```
public interface IFaxContent {
    boolean faxContent(String content);
}
```

- Une interface pour la fonction d'impression en recto verso

```
public interface IPrintDuplex {
    boolean printDuplexContent(String content);
}
```

## Principe de substitution de Liskov (LSP)

- Les sous-types doivent être substituables à leurs types de base.(Robert C. Martin).





(Turing Price, 2008)

## Principe de substitution de Liskov (LSP)

- **Définition** : Si une propriété P est vraie pour une instance x d'un type T, alors cette propriété P doit rester vraie pour tout instance y d'un sous-type de T »
- Implications :
  - Le «contrat» défini par la classe de base (pour chacune de ses méthodes) doit être respecté par les classe dérivées
  - L'appelant n'a pas à connaître le type exact de la classe qu'il manipule : n'importe quelle classe dérivée peut être substituée à la classe qu'il utilise
- → Principe de base du polymorphisme :
  - Si on substitue une classe par une autre dérivée de la même hiérarchie: comportement (bien sûr) différent mais conforme

## Principe de substitution de Liskov (LSP)

- Les sous-types doivent être substituables à leurs types de base.
- N'exigez pas plus, ne promettez rien de moins
  - n'exigez pas plus: la sous-classe accepterait tous les arguments que la superclasse accepterait.
  - ne promettez rien de moins: toute hypothèse valable lorsque la superclasse est utilisée doit l'être lorsque la sous-classe est utilisée.
- Héritage d'interface - Le LSP doit être conforme.
- Héritage d'implémentation - utilisez la composition au lieu de l'héritage (en Java).

## Principe de substitution de Liskov (LSP)

- Exemple (1/4):
  - Supposons la classe de base Employe qui proposent deux méthodes :

```
public abstract class Employe {
      protected String name;
      protected int ID;
      protected double salary ;
public Employe(String name, int ID,double salary) {
            this.ID = ID;
            this.name = name;
            this.salary = salary;

      }

public abstract double calculateBonus();
public abstract double getBaseSalary();

}
```

## Principe de substitution de Liskov (LSP)

- Exemple (2/4):
  - Les classes dérivées : *PermantentEmploye*, *TomporaryEmploye* qui proposent des implémentations des méthodes *calculateBonus* et *getBaseSalary*

```
public class PermanentEmploye extends Employe  {
      public  PermanentEmploye(String name, int id, double salary) {
            super(name, id,salary);
      }
      public double calculateBonus() {
            return salary*0.1;
      }

      public double getBaseSalary() {
            return this.salary*1.1;
      }
}
```

## Principe de substitution de Liskov (LSP)

- Exemple (3/4):

```
public class TemporaryEmploye extends Employe {

public  TemporaryEmploye(String name, int id, double salary) {
      super(name, id,salary);
}
public double calculateBonus() {
      return this.salary*0.05;
}
public double getBaseSalary() {
      return this.salary*1.05;
}
```

## Principe de substitution de Liskov (LSP)

- Exemple (4/4)
  - Supposons qu'on a une classe *ContractEmploye* dérivé de Employe mais qui ne proposent pas d'implémentation de la méthode *calculateBonus* //règle métier

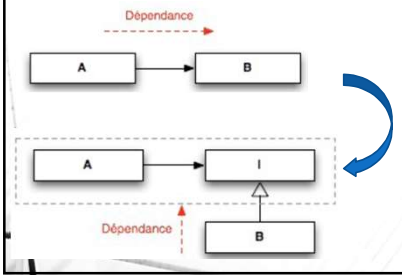
```
public class ContractEmploye  extends Employe {
      public ContractEmploye(String name, int id, double salary) {
            super(name, id,salary);
      }
      public double getBaseSalary() {
            return this.salary*1;
      }
      public double calculateBonus() {
            // Throw an exception
            // contract employe ne touche pas le bonus

      }
```

## Principe de l'inversion de dépendance

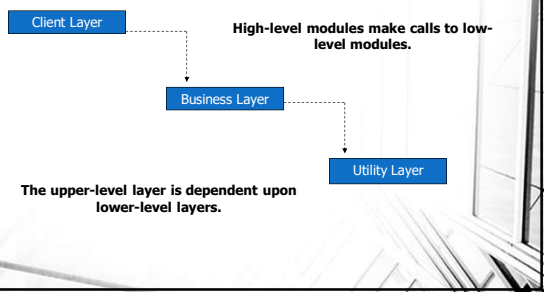
- Les abstractions ne doivent pas dépendre de détails. Les détails doivent dépendre d'abstraction. R. C. Martin
- Réduire les dépendances sur les classes concrètes
  - Program to interface, not implementation »
- Ne dépendre QUE des abstractions, y compris pour les classes de bas niveau
- Permet OCP (concept) quand l'inversion de dépendance c'est la technique!

## Principe de l'inversion de dépendance
### Technique

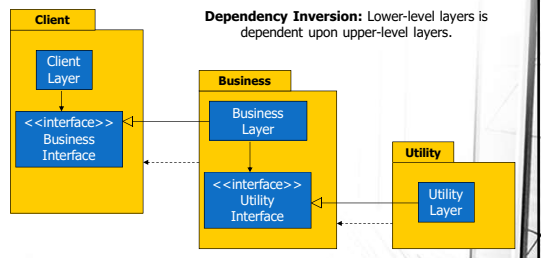
- Supposons que les classes A et B son reliées par une association dirigée du A vers B, on dit que A dépend de B.



## Principe de l'inversion de dépendance

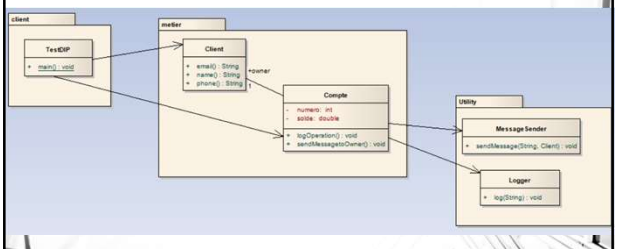



## Principe de l'inversion de dépendance




## Principe de l'inversion de dépendance

- Exemple



## Principe de l'inversion de dépendance

- Exemple (1/4): Classe client

```
public class Client {
    private int ID;
    private String name;
    private String email;
    public Client(int iD, String name, String email) {
        super();
        ID = iD;
        this.name = name;
        this.email = email;
    }
```

## Principe de l'inversion de dépendance

- Exemple (2/4) : classe Compte

```
public class Compte {
        private int number;
        private double solde;
        private Client owner;

        public Compte(int number, double solde, Client cli) {
            super();
            this.number = number;
            this.solde = solde;
            this.owner = cli;
        }
        public void logOperation() {
            Logger journal = new Logger();
            journal.log("compte created : "+this.getNumber());
        }

        public void sendMailtoOwner() {
            MailSender mailer = new MailSender();
            mailer.sendMail(this, "Solde = "+this.getSolde());
        }
```

## Principe de l'inversion de dépendance

- Exemple (3/4) : Les classes utilitaires : Logger et EmailSender

```
public class Logger {

    public void log(String message) {
        System.out.println(message);
    }
}
```

```
public class MailSender {
    public void sendMail(Compte cp, String message) {

        System.out.println("sending email to : "+cp.getOwner().getEmail()+ " , "  +message);
    };
}
```

## Principe de l'inversion de dépendance

- Exemple (4/4) : la classe Client de test

```
public class TestDIP {

    public static void main(String[] args) {

        Client cli = new Client(10,"Adil","anwar@emi.ac.ma");

        Compte cp = new Compte(7850,5000,cli);

        cp.logOperation();
        cp.sendMailtoOwner();
    }

}
```

## Résumé des principes SOLID

- **S**ingle-responsibility principle
  - There is only one source that may the class to change
- **O**pen-closed principle
  - Open to extension, closed for modification
- **L**iskov substitution principle
  - A subclass must substitutable for its base class
- **I**nterface segregation principle
  - A client should not be forced to depend on methods it does not use.
- **D**ependency inversion principle
  - Low-level (implementation, utility) classes should be dependent on high-level (conceptual, policy) classes

Ecole Mohammadia d'Ingénieurs

# Patrons de conception

**Pr : Adil ANWAR**
anwar@emi.ac.ma
**Année : 2019/2020**

1

# Plan

- Rappel des principes de base de la conception objet
- Principes S.O.L.I.D : écrire du bon code
  - Principe de responsabilité unique
  - Principe d'ouverture/fermeture
  - Principe de substitution de Liskov
  - Principe de séparation d'interfaces
  - Principe d'inversion de dépendances
- Design Patterns (GOF)

## Principes de la conception orientée objet

- les **principes de la conception orientée objet** (OOD) vous permettent de développer des applications bien conçues , maintenables, réutilisables et évolutives.
  - **Programmer pour des interfaces** : Limiter le couplage et protéger des variations en faisant abstraction de l'implémentation des objets
  - **Composer au lieu d'hériter** : Limiter le couplage en utilisant la composition (boite noire) au lieu de l'héritage (boite blanche) pour déléguer une tâche à un objet.
  - **Protection des variations** : Identifier les points de variation et d'évolution, et séparer ces aspects de ceux qui demeurent constants.
  - **Indirection** : Limiter le couplage et protéger des variations en ajoutant des objets intermédiaires (ex classes Façade, classes Factory)

## Patrons de conception

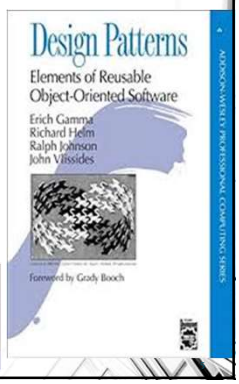
### Introduction

- Les gens qui vivent dans des régions qui ont des chutes de neige construisent des toits en pente afin que la neige et la glace ne s'accumule pas sur les toits.
- Ce «pattern» de conception de toits en pente a été identifié après l'expérience de plusieurs personnes qui ont vécu des difficultés similaires et ont trouvé des solutions similaires. Maintenant, c'est devenue une bonne pratique
- De même, dans le domaine informatique, de multiples design pattern ont été documentés en observant des problèmes récurrents de programmation, de comportement ou d'implémentation.

## Patrons de conception

- 1994 : **Erich Gamma**, **Richard Helm**, **Ralph Johnson**, et **John Vlissides** publièrent le livre

« Patrons de conception : Eléments de logiciels orienté-objet Réutilisable




## Patrons de conception

Catalogue

| | | |
|---|---|---|
| C Abstract Factory | S Facade | S Proxy |
| S Adapter | C Factory Method | B Observer |
| S Bridge | S Flyweight | C Singleton |
| C Builder | B Interpreter | B State |
| B Chain of Responsibility | B Iterator | B Strategy |
| B Command | B Mediator | B Template Method |
| S Composite | B Memento | B Visitor |
| S Decorator | C Prototype | |

## Patrons de conception

- Livre plus contemporain et plus pédagogique


7

## Patrons de conception

Que sont des patrons ?

- Un patron est un **schéma générique** d'une solution à un problème récurrent dans un **contexte** donné
- Ils ne sont pas inventés, mais découlent **d'expériences pratiques**.
- Ce sont des composantes qui doivent être **combinées** et **adaptées** pour résoudre des problèmes plus compliqués.
- Ils peuvent être utilisées comme **vocabulaire** pour communiquer à des niveaux **d'abstractions** plus élevés.
  - « c'est un objet composite » « il y'a un visiteur qui fait le calcul » « j'appelle une commande »
- Identifier les cibles du design à **restructurer** (refactoring) pour améliorer la **qualité** du logiciel et augmenter sa **réutilisation**

8

## Patrons de conception

**Pourquoi utiliser les patrons de conception ?**

- Parce qu'ils présentent des solutions testées et qui ont fait leurs preuves dans plusieurs cas similaires
  - Utilisés dans le passé avec succès
  - Une forme de réutilisation

**Les patrons sont pour la conception ce que les algorithmes et les structures de données sont pour le code**

9

## Patrons de conception

### Types de patrons de conception

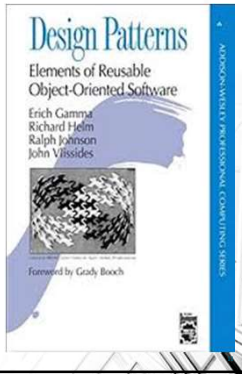
- **Patrons de construction**
  - Abstraire le processus d'instanciation.
  - Rendre le système indépendant de comment les objets sont créés, composés et représentées.
- **Patrons de structure**
  - Comment les classes et objets sont composés pour former de plus grandes structures.
  - Utilisées pour fournir de nouvelles fonctionnalités.
- **Patrons de comportement**
  - Algorithmes et **assignation** de responsabilités entre objets.
  - Communication entre objets et comportement interne de l'objet.

## Patrons de conception

### Catalogue

Quand vous êtes face à un problème de conception

- Chercher dans le catalogue si un patron de conception résout le problème.
- Avant d'implémenter la solution
  - Bien identifier les différents participants
  - Étudier rigoureusement le chapitre du livre, en particulier les sections des conséquences et implémentations.




## Patrons de conception

### Concevoir une classe qui utilise le design pattern Singleton

**Introduction au design pattern**

- Un design pattern identifie un problème spécifique et suggère une solution.
- Ce n'est ni du code prêt à utiliser, ni un framework à utiliser.
- Les designs patterns offrent une réutilisation de l'expérience et non la réutilisation du code
- Par exemple, vous pouvez documenter un design pattern de toit en pente comme suite :
  - ***Nom design pattern**: toit en pente*
  - ***Problème**: accumulation de neige et de glace sur les toits*
  - ***Solution suggérée**: Construire des toits en pente pour toutes les maisons, bureaux et bâtiments pour les zones qui reçoivent des chutes de neige pendant une période quelconque de l'année. Cela permet à la neige ou à la glace des toits glisser et tomber au sol.*

# Patrons de Création



## Patrons de conception

Concevoir une classe qui utilise le design pattern Singleton

**Design pattern Singleton**

- Singleton est un design pattern de création qui garantit qu'une classe est instanciée une seule fois
- Une instance **unique** de « Device Manager » pour gérer tous les périphériques de votre système et une instance **unique** d'un « spooler d'impression » pour gérer tous les travaux d'impression

## Patrons de conception

Concevoir une classe qui utilise le design pattern Singleton

**Design pattern Singleton**

**Implémentation**

- **Step 1** : Définir un constructeur privé pour la classe qui implémente le pattern Singleton.

  « Pour empêcher toute autre classe de créer un objet de cette classe, marquez le constructeur de cette classe en tant que méthode privé »

```
class Singleton {
    private Singleton() {
        System.out.println("Private Constructor");
    }
}
```

## Patrons de conception

Concevoir une classe qui utilise le design pattern Singleton

**Design pattern Singleton**

**Implémentation**

- **Step** 2 : Définir une variable **static private** pour désigner la seule instance de la classe Singleton.

  « Une variable statique garantit que la classe crée et accède à la même instance.»

```
class Singleton {
    private static Singleton anInstance = null;
    private Singleton() {
        System.out.println("Private Constructor");
    }
}
```

## Patrons de conception

Concevoir une classe qui utilise le design pattern Singleton

**Design pattern Singleton**

**Implémentation**

- **Step** 3 : Définir une méthode publique statique pour accéder à la seule instance de la classe Singleton.

  « Avant d'accéder à la variable **anInstance**, vous devez la créer. La création et le renvoi de cette variable est généralement défini comme suit.»

```
class Singleton {
    private static Singleton anInstance = null;
    public static Singleton getInstance() {
        if (anInstance == null)
            anInstance = new Singleton();
        return anInstance;
    }
    private Singleton() {
        System.out.println("Private Constructor");
    }
}
```

If anInstance hasn't been initialized

Initialize anInstance

Return anInstance

## Patrons de conception

Concevoir une classe qui utilise le design pattern Singleton

**Design pattern Singleton**

**Implémentation**

- **Step** 4 : Une classe peut demander un objet de classe Singleton en appelant la méthode statique.

```
class UseSingleton {
    public static void main(String args[]) {
        Singleton singleton1 = Singleton.getInstance();
        Singleton singleton2 = Singleton.getInstance();
        System.out.println(singleton1 == singleton2);
    }
}
```

Prints "true"

New instance of class Singleton created and returned when accessed first time

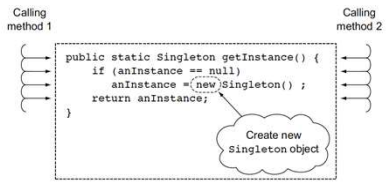
Previously created instance returned for subsequent calls to method getInstance()

## Patrons de conception

Concevoir une classe qui utilise le design pattern Singleton

**Design pattern Singleton**

**Multi-thread contrainte**




## Patrons de conception

Concevoir une classe qui utilise le design pattern Singleton

**Design pattern Singleton**

**Multi-thread Synchronized lazy initialization**

- **Synchronized lazy initialization** applique un verrou sur la méthode qui crée et renvoie un singleton.
- Elle optimisation de la performance avec un verou partiel.

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

Don't synchronize complete method

Synchronize code block that creates new object

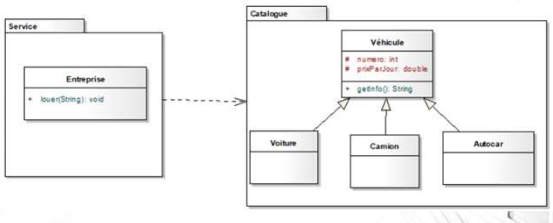
## Patrons de conception

Patrons de fabriques

- **Patrons de création qui permettent de :**
  - Dissimuler comment créer les objets
  - Permet au code d'être indépendant de comment les objets sont créés, composés et représentés.
  - Les objets sont créés sans exposer la logique d'instanciation au client.
  - L'application cliente est découplée de l'implémentation de la classe instanciée.
- **Plusieurs modèles de patrons de fabriques:**
  - Méthode de fabrication (Factory Method)
    - Délègue l'instanciation à une sous-classe
  - Fabrique abstraite (Abstract Factory)
    - Délègue l'instanciation à un autre objet

## Patrons de conception
### Patrons de fabriques

- Exemple d'introduction




## Patrons de conception
### Patrons de fabriques

- Quels sont les problèmes avec cette conception ?

```
public class Entreprise {
    public Vehicule louer(String type) {
        Vehicule v=null;
        if(type.equalsIgnoreCase("voiture"))
            v = new Voiture();
        else if(type.equalsIgnoreCase("camion"))
            v = new Camion();
        else if(type.equalsIgnoreCase("autocar"))
            v = new AutoCar();

        return v;
    }
}
```

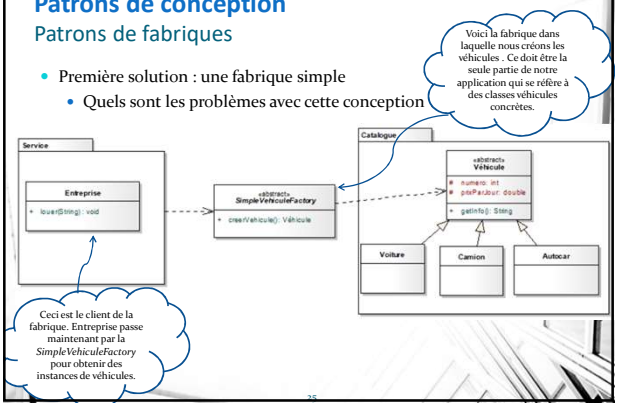
- Pas de DIP, pas d'OCP : identifier les points de variations…
- l'obligation de savoir quelle classe concrète est instanciée bousille la méthode louer() et l'empêche d'être fermée à la modification.

## Patrons de conception
### Patrons de fabriques

- Première solution : une fabrique simple
  - nous allons prendre le code de la méthode louer() et le placer dans un autre objet qui ne s'occupera que d'une seule chose : créer des véhicules.
- Les Fabriques gèrent les détails de la création des objets. Une fois que nous avons une SimpleFabriqueDeVehicule, notre méthode louer() devient simplement un client de cet objet. Chaque fois qu'elle aura besoin d'un véhicule,
- elle demandera à la fabrique de véhicules de lui en faire un. Le temps n'est plus où la méthode initialiser() devait savoir si le véhicule était une voiture ou un camion. Maintenant, une seule chose lui importe : obtenir un véhicule.

## Patrons de conception
### Patrons de fabriques

- Première solution : une fabrique simple
  - Quels sont les problèmes avec cette conception





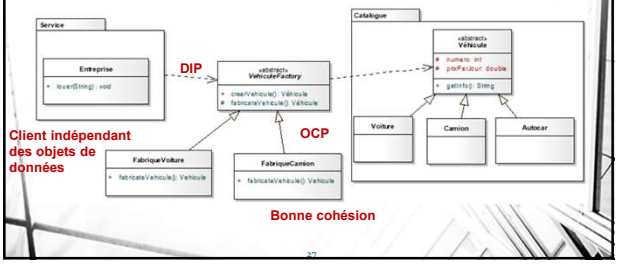

## Patrons de conception
### Patrons de fabriques

- Vers les vrais patrons de fabrication ....
  - La fabrique Simple n'est pas vraiment un Design Pattern : c'est plutôt un principe de conception objet→ **Indirection** : **Limiter le couplage** et **protéger des variations** en ajoutant des objets intermédiaires (ex classes Factory, Façade)
  - Mais il existe encore des problèmes
    - Encore trop de dépendances avec des classes concrètes
    - Il-else encore là
  - Considérez Fabrique Simple comme un échauffement. Nous allons bientôt explorer deux patterns très robustes qui sont tous deux des fabriques.



## Patrons de conception
### Patrons de fabriques

- Solution : Satisfaction des principes
  - Déléguer le choix aux sous classes

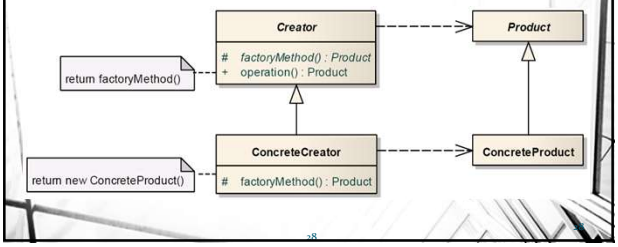





fabriqueVehicule() est abstract, c'est elle qui crée les sous class Voiture et camion a l'aide de FabriquVoiture et FabriqueCamion, par contre creerVehicule() va être utliser par fabriqueVehicule() pour créer la Classe Véhicule et qui soit appelée de l'extérieur par Entreprise.

## Patrons de conception
Méthode de fabrication ou factory method

- Structure :
  - Définir une **interface** pour créer un objet, mais en laissant les **sous-classes** décider quelle classe instancier.

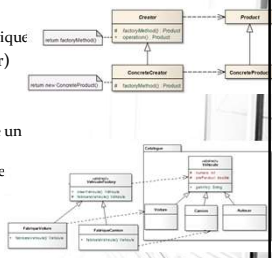





## Patrons de conception
Méthode de fabrication ou factory method

Participants

- Product (Vehicule)
  - Définit l'interface des objets crées par la fabrique
- ConcreteProduct (voiture, camion, autocar)
  - Implémente l'inteface product
- Creator (VehiculeFactory)
  - Déclare la fabrique de méthode qui retourne un objet product
  - Appelle la méthode de fabrique pour créer de produits
- ConcreteCreator (FabriqueVoiture, FabriqueCamion)
  - Redéfinition (override) la méthode de fabrication pour retourner une instance de concreteProduct



fabriquerVehicule() rôle interne : savoir quel type concret d'objet créer.

creerVehicule() rôle externe : fournir un produit prêt à être utilisé par Entreprise.

## Patrons de conception
Méthode de fabrication ou factory method

Commentaires

- Encapsule le savoir de quel produit concret il faut créer et élimine cette dépendance du reste de l'application.
  - Dépend seulement de l'interface produit : fonctionne avec n'importe quelle classe produit concret.
- Réduit, la dépendance aux paquetages concrets
  - Créer une instance d'un type qui réside dans un paquetage différent du créateur augmente le couplage du paquetage
  - Référencer à un objet de fabrique pour créer des instances réduit ce couplage
- Hiérarchie de classes parallèles
  - Pour les créateurs et les produits
  - doit sous-classer créateur pour un produit spécifique

## Patrons de conception
### Patrons de fabriques

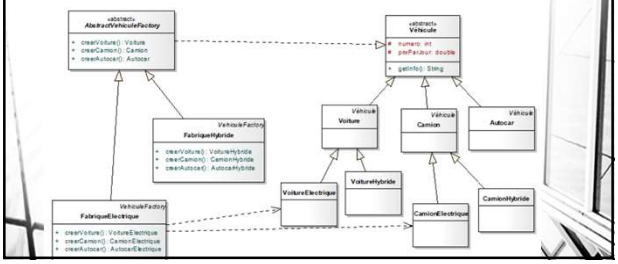
Changement d'exigences

- le marché des véhicules évolue à grande vitesse et la tendance actuelle s'oriente vers les voitures électriques et hybrides
- Question : comment peut-on ajouter de nouveaux types produits :
  - Chaque véhicule peut être de type électrique, hybride, essence, gazoil
  - Comment peut on **créer** une famille de produit de même type? un catalogue de véhicules hybrides, électriques, etc.?

31

## Patrons de conception
### Fabrique Abstraite Abstract Factory

- Offre une **interface** pour créer des **familles** d'objets reliés ou indépendants sans spécifier leurs classes concrètes




## Patrons de conception
### Fabrique Abstraite Abstract Factory

Quand utiliser ce patron ?

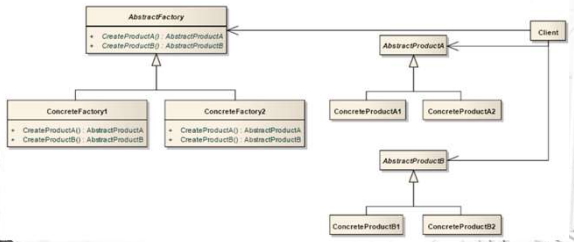
- Code doit être *indépendant* de comment les **produits** sont crées
- La classe ne peut pas **anticiper** la classe des objets à créer
- Le client utilise seulement **une** des **familles** de produits
- Une famille de produits reliés sont conçues de sorte qu'ils doivent être **utilisés ensemble.**
- Offrir une **librairie de classes** de produits, mais en révelant seulement leur **interface**, pas leur **implémentation**.

33

## Patrons de conception

Fabrique Abstraite Abstract Factory

Structure




34

## Patrons de Structure

35

## Patrons de conception

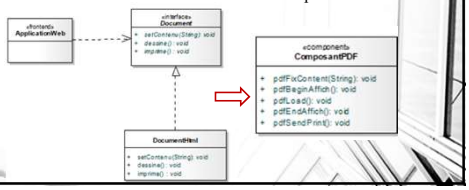
Patrons de structure

- Les Patrons de conception tentent de composer des classes pour bâtir de nouvelles structures.
- Ces structures servent avant tout à ne pas gérer différemment des groupes d'objets et des objets uniques.
- Autrement dit, l'objectif est de faciliter l'indépendance de l'interface d'un objet ou d'un ensemble d'objet vis-à-vis de son implantation.
  - Dans le cas d'un ensemble d'objets, il s'agit aussi de rendre cette interface indépendante de la hiérarchie des classes et de la composition des objets.
- Par exemple : en utilisant un logiciel de dessin vectoriel, tout le monde est amené à grouper des objets.
  - Les objets ainsi conçus forment un nouvel objet que l'on peut déplacer, et manipuler sans avoir à répéter ces opérations sur chaque objet qui le compose.
  - On obtient donc une structure plus large mais toujours facilement manipulable.

## Patrons de conception

### Patrons de structure

- Problème
  - Une application web d'un système de vente de véhicules créée et gère des documents destinées aux clients.
  - L'interface *Document* a été définie pour cette gestion. Cette interface contient trois méthodes *setContenu*, *dessine* et *imprime*.
  - Une première classe d'implémentation de cette interface a été réalisée : la classe *DocumentHTML* qui implante ces trois méthodes. Les objets clients de cette interface et de cette classe ont été conçus.
  - Changement des exigences, l'ajout des documents PDF. Un composant du marché à été choisi mais dont l'interface ne correspond à l'interface *Document*.




## Patrons de conception

### Patron adaptateur

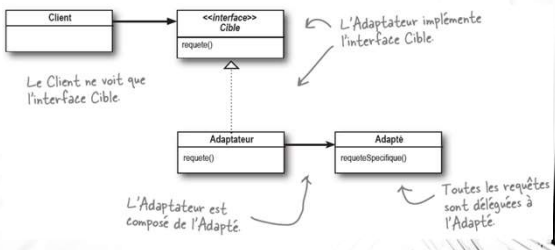
- **Problème**
  - Communication entre deux objets ayant leurs implémentations **incompatibles**
- **Solution**
  - **Convertir l'interface** d'une classe en une autre interface attendue par le client afin de leur permettre de travailler ensemble.
- Donne accès à son implémentation interne sans pour autant **coupler** les clients à son implémentation interne.
- Il s'agit de conférer à une classe existante une nouvelle interface pour répondre aux besoins de clients.
- Adaptateur fournit tous les avantages de la **dissimulation de l'information** sans avoir à cacher ses détails d'implémentation

38

## Patrons de conception

### Patron adaptateur

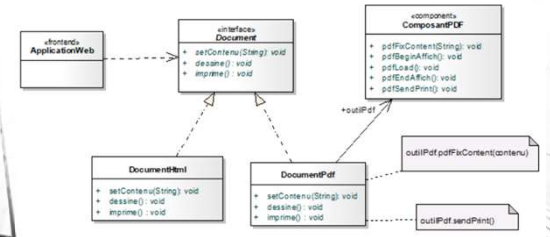
Solution générique




39

## Patrons de conception
Patron adaptateur

Solution du problème




40

## Patrons de conception
Patron adaptateur

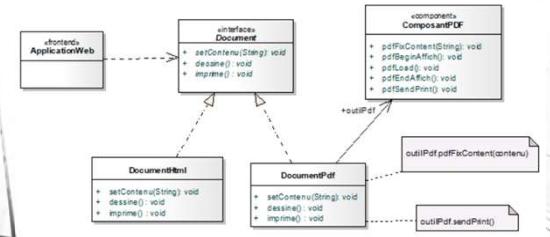
Participants

- Les participants au pattern sont les suivants :
  - **Cible** (Document) : introduit la signature des méthodes de l'objet.
  - **Client** (Application web) : interagit avec les objets répondant à l'interface cible.
  - **Adaptateur** (DocumentPdf) : implante les méthodes de l'interface cible en invoquant les méthodes de l'objet adapté.
  - **Adapté** (ComposantPdf) : introduit l'objet dont l'interface doit être adaptée pour correspondre à l'interface cible.

41

## Patrons de conception
Patron adaptateur

Solution du problème

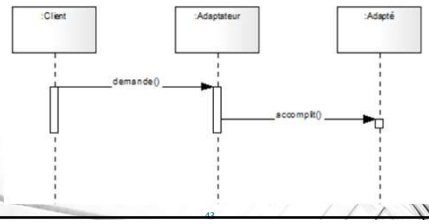



42

## Patrons de conception
Patron adaptateur

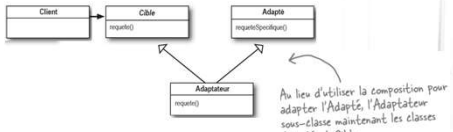
### Collaborations

- Le client invoque la méthode *demande* de l'adaptateur qui, en conséquence, interagit avec l'objet *Adapté* en appelant la méthode *accomplit*.




## Patrons de conception
Patron adaptateur

### Adaptateur d'objets vs adaptateur de classe




## Patrons de conception
Patron adaptateur

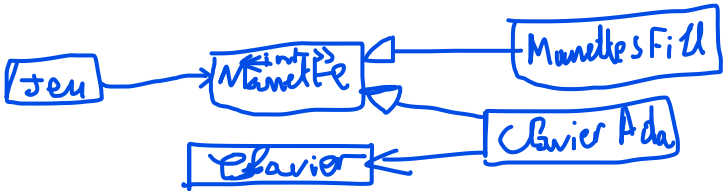
### Adaptateur d'objets vs adaptateur de classe

**Objet**

- doit adapater toutes les méthodes
- utilise la délégation pour adapter toutes les sous-classes de l'adapté
- Plus de flexibilité

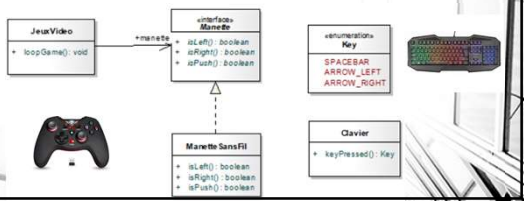
**Classe**

- Peut override uniquement les méthodes à adapter
- Plus de performance




## Patrons de conception
### Patron adaptateur

- Exercice
  - Nous avons développer un jeu vidéo. Dans une première version, le jeu a été conçu pour être jouable à la manette
  - Nous désirons mettre à jour le jeu pour permettre également le jeu avec un clavier, avec le minimum d'impact sur le code existant

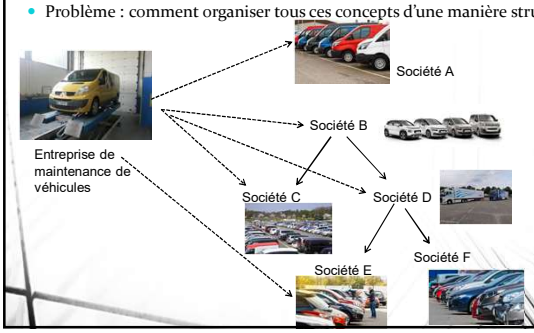



## Patrons de conception
### Patron composite

- Problème
  - Une entreprise de maintenance possède un système de gestion de véhicules, les clients cette entreprise sont particulièrement des sociétés.
  - il existe deux catégories de sociétés, il y'a les sociétés qui possédant des filiales et les sociétés sans filiales. Notons, que les sociétés qui possèdent des filiales et qui peuvent posséder elles mêmes d'autres filiales.
  - l'entreprise de maintenance souhaite géré les deux catégories de sociétés d'une façon transparente (connaitre le nombre de véhicules, Coût de la maintenance, etc.).
  - Les deux types de sociétés (avec ou sans) filiales demandent des offres de maintenances qui prennent en compte le parc de véhicules de leurs filiales.

## Patrons de conception
### Patron composite

- Problème : comment organiser tous ces concepts d'une manière structurée?



## Patrons de conception
### Patron composite

**Hiérarchie en arborescence**

- Les éléments sont placés dans une structure hiérarchique.
- Manipuler les nœuds composites et primitifs de la même manière.
  - Calcul du coût de maintenance
  - Ajout, suppression de véhicules
- Appliquer des traitements sur une structure de données « arbre » sans distinguer les feuilles des nœuds.
- Le client n'a pas à se soucier du type d'objet (société mère, sans filiale) avec lequel il fait affaire.
- Tout composant doit implémenter la méthode de calcul du coût
  - Même interface



## Patrons de conception
### Patron composite

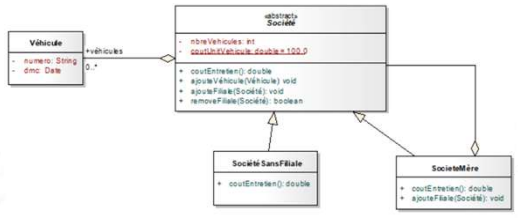
**Objet Composite**

- Compose les objets en structures d'arborescence pour représenter des hiérarchies d'ensemble/partie.
- Permet au client de traiter les objets et les compositions de façon uniforme.
  - Objets primitifs, atomiques
  - Objets composites.



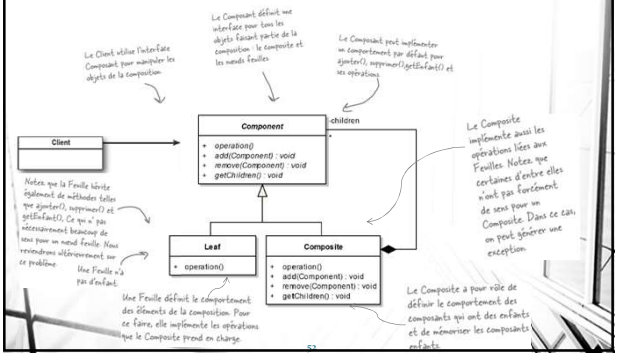
## Patrons de conception
### Patron composite

- Solution : diagramme de classes



## Patrons de conception

Patron composite : Structure




## Patrons de conception

Patron composite

Participants

- **Composant**
  - Déclare l'interface pour les objets
  - Implémente le comportement par défaut.
  - Déclare les interfaces de gestion des composants enfants
- **Feuille**
  - Représente les primitives : pas d'enfants
- **Composite**
  - Définit le comportement des composants qui ont des enfants
  - Implémente les opérations de gestion des enfants
- **Client**
  - Manipule les objets via l'interface composant

## Patrons de conception

Patron composite

- Implémentation : Composant abstrait

```
public abstract class Societe {
    protected String name;
    protected double coutUnitVehiucle=1000.0;
    protected int nbreVehicule;
    protected List<Vehicule> flotte = new ArrayList<Vehicule>();

    public abstract void addFiliale(Societe s);
    public abstract void removeFiliale(Societe s);
    public abstract double coutEntretien();

    public Societe(String name) {
        this.name = name;
    }
      public  void addVehicule(Vehicule s) {
            this.nbreVehicule++;
        }
       public void removeVehicule() {
            if(this.flotte.size() > 1)
                this.nbreVehicule--;
        };
}
```

## Principes de la conception orientée objet
Patron composite

- Implémentation : Composant

```
public class SocieteSansFiliale extends Societe {
    public SocieteSansFiliale(String name) {
        super(name);
    }
    public void addFiliale(Societe s) {
        throw new UnsupportedOperationException("unsupported add filiale operation ")
    }
    public void removeFiliale(Societe s) {
        throw new UnsupportedOperationException("unsupported remove filiale operation
    }
    public double coutEntretien() {
        return this.coutUnitVehiucle*this.nbreVehicule;
    }
}
```

55

## Patrons de conception
Patron composite

- Implémentation : composite

```
public class SocieteMere extends Societe {
     protected List<Societe> filiales = new ArrayList<Societe>();
    public SocieteMere(String name) {
        super(name);
    }
    public void addFiliale(Societe s) {
        this.filiales.add(s);
    }
    public void removeFiliale(Societe s) {
        if(this.filiales.contains(s))
            this.filiales.remove(s);
    }
    public double coutEntretien() {
        double cout=0.0;
        for(Societe s : this.filiales)
            cout=cout+s.coutEntretien();
        return cout+this.coutUnitVehiucle*this.nbreVehicule;
    }
}
```

56

## Patrons de conception
Patron composite

- Implémentation : Classe de test

```
public class EntrepriseMaintenance {

    public static void main(String[] args) {
        Societe groupe1 = createTreeOne();
        System.out.println("coût d'entretien du groupe1 : "+groupe1.coutEntretien());
        System.out.println("************************************");

        Societe groupe2 = createTreeTwo();
        System.out.println("coût d'entretien du groupe2 : "+groupe2.coutEntretien());
    }
  private static Societe createTreeOne() {
      Societe groupe = new SocieteMere("Carrefour");
        groupe.addVehicule(new Vehicule("4750NE",LocalDate.now()));
        Societe societe1 = new SocieteSansFiliale("Carrefour Market");
        Societe societe2 = new SocieteSansFiliale("Carrefour géant");
        societe1.addVehicule(new Vehicule("754CV",LocalDate.now()));
        societe2.addVehicule(new Vehicule("7824VR",LocalDate.now()));
        groupe.addFiliale(societe2);
        groupe.addFiliale(societe1);
        return groupe;
    }
```

57

## Patrons de conception
### Patron composite

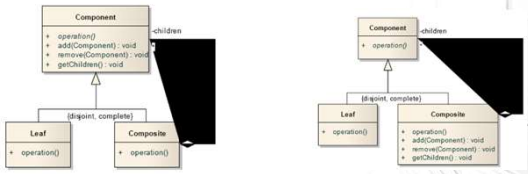
*Opérations inutilisées*

- Composant est abstrait, donc toutes les primitives doivent implémenter les opérations ajouter() et supprimer().
- Est-ce que les primitives ont vraiment besoin de ces méthodes ?
- Solution facile : signaler une exception quand ces méthodes sont appelées.
  - Mauvais design : **violation du LSP** + cohésion d'instance mixte.
- Où déclarer les méthodes ajouter() et supprimer() ?
  - Dans Composant ?
    - Laisse des méthodes insensées
  - Dans composite ?
    - Brise l'abstraction



## Patrons de conception
### Patron composite

**Compromis**
**opération de gestion des enfants**



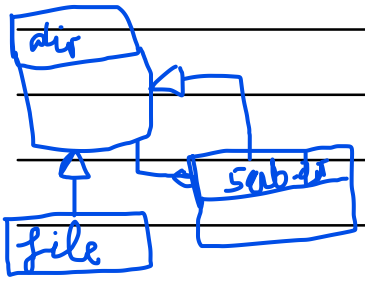

Transparence vs Sécurité



## Patrons de conception
### Patron composite

- Exercice
  - Représenter par un diagramme de classes le système de gestion de fichiers suivant (vu par un utilisateur) :
    - les fichiers, les répertoires sont contenus dans des répertoires et possèdent un nom et une taille et une liste de permissions (R, W et X);
    - au sein d'un répertoire donné, un nom ne peut identifier qu'un seul élément (fichier, sous-répertoire).
    - les fichiers et les répertoires possèdent une méthode ls() permettant d'afficher le nom du fichier ou du sous-répertoire suivi de la taille et de la liste de permissions.
  - Créer une application java pour implémenter la solution proposée et un classe de test qui créer et affiche la structure de deux arbres différents.

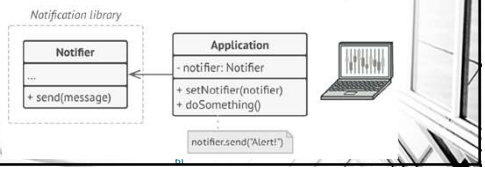




## Patrons de conception

### Patrons Structuraux

- Problème: gestion des notifications
  - On souhaite développer un programme de gestion des notifications d'une bibliothèque, ce système permet d'informer les utilisateurs de la bibliothèque par email des événements importants (nouveaux arrivages de livres, etc.).
  - Suite à un changement d'exigences le programme à évolué pour prendre en compte plusieurs types de notificateurs (SMS, facebook, etc).

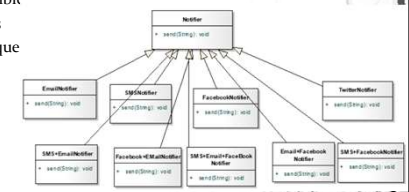



61

## Patrons de conception

### Patrons Structuraux

- 1ère **Solution** : héritage
- Supporter toutes les variantes
  - OCP satisfait…mais
  - Maintenance impossible
  - Explosion de classes
  - Répétition de la logique




62

## Patrons de conception

### Patrons Structuraux

- 1ème solution : critiques

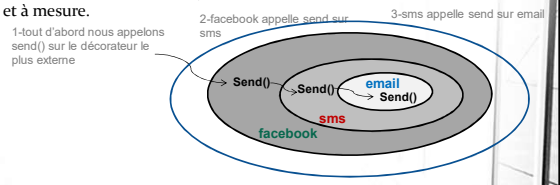
```
public class Notifier {

    private boolean email,sms,facebook = false;
    boolean hasSMS() {return sms;}
    boolean hasFaceBook() {return facebook;}
    protected  void send(String message){
        System.out.println("send email to client : "+message);
        if(hasSMS())
            System.out.println("send facebook message to client : "+message);
        if(hasFaceBook())
            System.out.println("send sms to client : "+message);

    }
}
```

63

## Patrons de conception
Patron Décorateur

**Solution : Décorateur**

- Décorer ou envelopper l'objet final avec les canaux de notifications au fur et à mesure.




- Ajouter des responsabilités à des objets dynamiquement. Cet ajout de responsabilités ne modifie pas l'interface de l'objet et reste donc transparent vis-à-vis des clients.

64

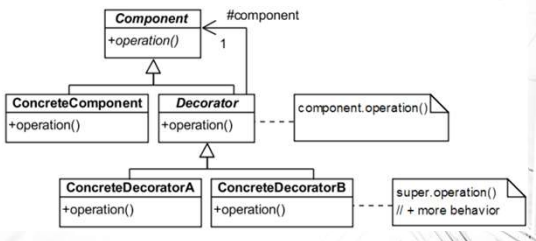
## Patrons de conception
Patron Décorateur

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

65

## Patrons de conception
Patron Décorateur

**Structure**




66

## Patrons de conception
### Patron Décorateur

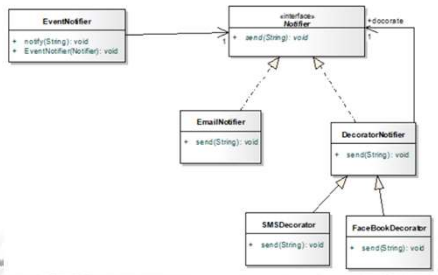
**Participants**

- Composant (*Notifier*)
  - Déclare l'interface des objets auxquels on peut ajouter des responsabilités dynamiquement.
- Composant concret (*EMailNotifier*)
  - Définit un tel objet
- Décorateur (*DecoratorNotifier*)
  - **Maintien la référence** à un composant et définit une interface qui lui est conforme
  - Délègue la requête au composant
- Décorateur Concret (*SMSDecorator* et *FacebookDecorator*, *TwiterDecorator*)
  - Encapsule l'ajout d'une responsabilité du composant.

67

## Patrons de conception
### Patron Décorateur

Décorateur pour la gestion des notifications




68

## Patrons de conception
### Patron Décorateur

En Action

Scénario de notification avec facebook, sms et un email

(diagramme de séquences)

69

## Patrons de conception
### Patron Décorateur

Implémentation

```
public interface INotifier {
  void send(String message);
}
```

```
public class EmailNotifier implements INotifier {
    public EmailNotifier() {
    }
    public void send(String message) {
        System.out.println("send email to client : "+message);
    }
}
```

```
public abstract class DocoratorNotifier implements INotifier {
    INotifier decorated;
    public DocoratorNotifier(INotifier decorated) {
        this.decorated = decorated;
    }
    public abstract void send(String message);
}
public class SMSDecorator extends DocoratorNotifier {
    public SMSDecorator(INotifier decorated) {
        super(decorated);
    }
    public void send(String message) {
        System.out.println("send sms to client : "+message);
        this.decorated.send(message);
    }
}
```

## Patrons de conception
### Patron Décorateur

Implémentation

```
public class ClientDecorator {

    public static void main(String[] args) {
        EventNotificator event = new EventNotificator(new EmailNotifier());
        event.notify("arrivage du livre design patterns");

        INotifier notifier1 = new EmailNotifier();
        notifier1 = new SMSDecorator(notifier1);
        event.setNotifier(notifier1);
        event.notify("arrivage du livre design patterns");

        INotifier notifier2 = new EmailNotifier();
        notifier2 = new SMSDecorator(notifier2);
        notifier2 = new FaceBookDecorator(notifier2);
        event.setNotifier(notifier2);
        event.notify("arrivage du livre design patterns");
    }
```

Créer un événement avec une notification par défaut avec un email

Créer un objet email

L'envelopper dans un sms

Créer un objet email

L'envelopper de sms

L'envelopper de facebook

Enfin nous notifions le client avec email, un sms et un message facebook

## Patrons de conception
### Patron Décorateur

Conséquences

- Plus de flexibilité qu'avec l'héritage statique
  - Peut ajouter et d'enlever des responsabilités dynamiquement (SRP)
  - Pas besoin de créer une nouvelle classe pour chaque responsabilité (OCP) et d'ajouter la même propriété plusieurs fois (DRY).
- Prévient d'avoir une classe surchargée de propriétés à la base de la hiérarchie
  - Pas besoin de connaître toutes les fonctionnalités dés le début
    - Ajoutés à la demande
    - **Meilleure cohésion**

## Patrons de conception
Patron Décorateur

- Exercice
  - Voir TD GOF



## Patrons de Comportement



## Patrons de conception
Patrons de comportement

- Les Patrons de comportement proposent des solutions pour organiser les **interactions** et pour répartir les **responsabilités** entre les objets
- D'une manière générale, un patron de comportement permet de réduire la complexité de la communication entre objets et aussi la gestion du comportement interne de l'objet.
- nous allons étudier trois patrons dans cette catégorie :
  - Stratégie (Strategy)
  - Observateur (Observer)
  - Patron de méthode (Template Method)

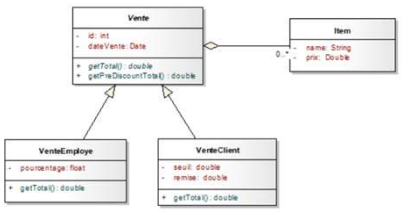
## Patrons de conception
### Patrons de comportement

- Problème
  - On considère la classe Vente qui est définie par :
    - une méthode getPreDiscountTotal qui retourne le montant total de la vente avant la remise.
    - Une méthode *getTotal* qui retoure le montant total après la remise.
  - Supposant que la formule de calcul de la remise diffère en fonction de plusieurs paramètres. L'idée est de fournir une logique de remise plus complexe, comme une remise du magasin pour une journée donnée, des remises les employés (pourcentage), une remise pour les clients si le montant dépasse un seuil donnée, etc.
  - La stratégie de remise (qui peut également être appelée règle, politique ou algorithme) pour une vente peut varier. Pendant une période, il peut être de 10% sur toutes les ventes, plus tard, il peut être de 50 dh. Si le total de la vente est supérieur à 2000 dh



## Patrons de conception
### Patrons de comportement

- 1ère Solution : Utilisation de l'héritage
  - Quels sont les problèmes de cette solution ?






## Patrons de conception
### Patrons de comportement

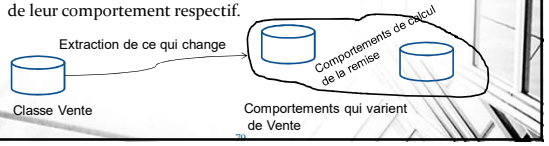
Attaquons le problème

**Principe de conception**
Identifier les aspects qui changent dans votre application et séparez-les de ceux qui demeurent constants.

1er Principe de conception

## Patrons de conception
### Patrons de comportement

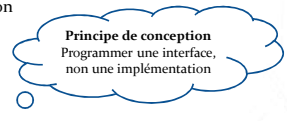
- Séparons ce qui change de ce qui reste identique.
  - Pour l'instant, en dehors du problème de getTotal() la classe Vente fonctionne bien et ne contient rien qui semble changer fréquemment. Nous allons la laisser pratiquement telle quelle.
  - Maintenant, pour séparer les parties qui changent de celles qui restent identiques, nous allons créer deux ensembles de classes, totalement distincts de Vente, l'un pour les clients et l'autre pour employés.
  - Chaque ensemble de classes contiendra toutes les implémentations de leur comportement respectif.




79

## Patrons de conception
### Patrons de comportement

- Comment allons-nous procéder pour concevoir les classes qui implémentent les deux types de comportements?
  - Pour conserver une certaine souplesse, nous allons procéder de façon que nous puissions affecter un comportement de calcul de la remise de manière dynamique à un objet Vente créé.
  - Avec ces objectifs en tête, voyons notre deuxième principe de conception

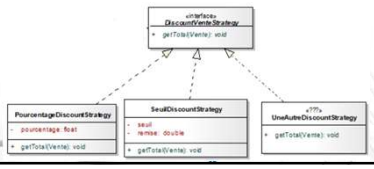



- 2ème Principe de conception

80

## Patrons de conception
### Patrons de comportement

- Conception du comportement de la classe vente
  - Nous allons donc utiliser une interface *RemiseVenteStrategy* pour représenter le comportement de calcul de la remise
  - Cette fois, ce n'est pas la classe Vente qui implémenteront cette interface, mais nous allons créer un ensemble de classes qui implémenteront cette interface et dont la seule raison d'être est de représenter un comportement.



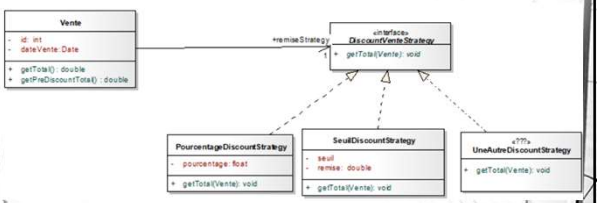
## Patrons de conception
Patrons de comportement

- 2ème Solution
  - Puisque le comportement de la remise varie selon la stratégie (ou l'algorithme), nous créons plusieurs classes implémentant l'interface *RemiseVenteStrategy*, chacune avec une méthode getTotal
  - La méthode getTotal prend l'objet Vente comme paramètre,
  - l'objet de la stratégie de remise peut utiliser le prix avant remise sur la vente, puis appliquez la règle de remise.
  - La mise en œuvre de chaque getTotal sera différente: RemisePourcentageStrategy réduira d'un pourcentage, RemiseSeuilStratégie réduire le prix si le montant dépasse un seuil donnée, etc.



## Patrons de conception
Patrons de comportement

- Solution






## Patrons de conception
Patrons de comportement

- Avec cette conception, les objets instances de vente peuvent réutiliser nos comportements de calcul de la remise parce que ces comportements ne sont pas cachés dans la classe Vente!
  - Et nous pouvions ajouter de nouveaux comportements sans modifier aucune des classes comportementales existantes, OCP vérifié
  - Nous obtenons ainsi les avantages de la réutilisation sans la surcharge qui accompagne l'héritage.
  - La clé est qu'un objet Vente va maintenant **déléguer** son comportement au lieu d'utiliser la méthode *getTotal* définie dans la classe Vente.
  - la classe Vente a une variable d'instance nommées: remiseStrategy
    - Cet attribut sera déclaré de type d'interface et non de type d'implémentation concrète.
  - Chaque objet Vente affectera à cette variable de manière polymorphe pour référencer le type de comportement spécifique qu'il aimerait avoir à l'exécution (RemiseAvecSeuil, RemisePourcentage, etc...)

## Patrons de conception
### Patrons de comportement

La composition peut être préférable à l'héritage

- Chaque Vente a une stratégie de calcul de la remise auquel il délègue le comportement à un objet implémentant l'interface remise.
- Lorsque vous avez deux classes de la sorte, vous utilisez le composition.
  - Au lieu d'hériter leur comportement, les objets instances de vente l'obtiennent en étant composés avec le bon objet comportemental.
  - Cette technique est importante; en fait, nous avons appliqué notre troisième principe de conception:

**Principe de conception**
Préférez la composition à l'héritage

3 ème Principe de conception

85

## Patrons de conception
### Patrons de comportement

vous venez d'appliquez le design pattern appelé
« STRATEGIE »

- Le pattern Stratégie définie une **famille d'algorithmes**, **encapsule** chacun d'eux et les rend **interchangeables**.
- Le pattern Stratégie permet à l'algorithme de **varier** indépendamment des **clients** qui l'utilise.
- Grace à la composition ce pattern procure beaucoup de souplesse.
- Et surtout, vous pouvez **modifier** le **comportement** au moment de **l'exécution** tant que l'objet avec lequel vous composez implémente la bonne interface comportementale.

86

## Patrons de conception
### Patron Stratégie

- Implémentation de l'exemple

```
public  class Vente {
    private int id;
    private LocalDate date = LocalDate.now();
    private Map<String, Double> items= new HashMap<String, Double>();
    private DiscountVenteStrategy discoutStrategy;

    public Vente(DiscountVenteStrategy discoutStrategy) {
        super();
        this.discoutStrategy = discoutStrategy;
    }
    public double getPreDiscountTotal() {
        double pdt=0.0;
        for(Double value : items.values()) {
            pdt+=value;
        }
        return pdt;
    }

    public double getTotal() {
        return discoutStrategy.getTotal(this);
    }
```

87

## Patrons de conception
Patron Stratégie

- Implémentation de l'exemple
  - L'interface *DiscountVenteStrategy*

```
public interface DiscountVenteStrategy {
    double getTotal(Vente vente);
}
```

  - *L'implémentation de l'interface*

```
public class PourcentageDiscountStrategy implements DiscountVenteStrategy {
    private double pourcentage;

    public PourcentageDiscountStrategy(double pourcentage) {
        this.pourcentage = pourcentage;
    }
    public double getTotal(Vente vente) {
        return vente.getPreDiscountTotal()-vente.getPreDiscountTotal()*pourcentage;
    }
}
```

88

## Patrons de conception
Patron Stratégie

- Implémentation de l'exemple
  - une deuxième implémentation de l'interface

```
public class PalierDiscountStrategy implements DiscountVenteStrategy {
    private double discount;
    private double seuil;

    public PalierDiscountStrategy(double seuil, double discount) {
        this.discount = discount;
        this.seuil = seuil;
    }
    public double getTotal(Vente vente) {
        double pdt = vente.getPreDiscountTotal();
            if ( pdt < seuil )
                return pdt;
            else
                return pdt - discount;
    }
}
```

89

## Patrons de conception
Patron Stratégie

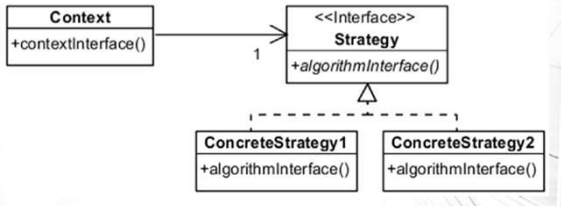
- Implémentation de l'exemple
  - Une classe Client de test

```
Detail de la vente :
Vente id #78420
Articles
-------------------------
Console XBox one          3500.8 dh
Pc portable HP probook    8000.8 dh
Sumsung Smart TV 49       4900.9 dh
-------------------------
total avant la remise     16402.5
-------------------------
total                     15582.375
```

```
public class ClientStrategy {
    public static void main(String[] args) {
        Vente vente1 = new Vente(new PourcentageDiscountStrategy(0.05));
        vente1.setId(78420);
        vente1.addItem("Sumsung Smart TV 49", 4900.90);
        vente1.addItem("Pc portable HP probook", 8000.80);
        vente1.addItem("Console XBox one", 3500.80);
        System.out.println("Detail vente :");
        System.out.println("vente id #"+vente1.getId());
        System.out.println("Articles\n-------------------------");
        for(Map.Entry<String, Double> item : vente1.getItems().entrySet()) {
            System.out.println(item.getKey() + " \t"+item.getValue()+" dh");
        }
        System.out.println("-------------------------");
        System.out.println("total avant la remise \t\t"+vente1.getPreDiscountTotal());
        System.out.println("-------------------------");
        System.out.println("total\t\t\t"+vente1.getTotal());

    }
}
```

## Patrons de conception
Patron Stratégie

Structure




## Patrons de conception
Patron Stratégie

Participants

- **Stratégie (DisocuntVenteStrategy)**
  - Déclare l'interface commune pour tous les algorithmes
  - Contexte utilise cette interface
- **Stratégie concrète (PourcentageDiscountStrategy, SeuilDiscountStrategy)**
  - Implémente l'algorithme
- **Contexte (Vente)**
  - fortement couplé avec la stratégie
  - Peut offrir une interface pour laisser la stratégie accéder à ses données.

## Patrons de conception
Patron Stratégie

- Exercice
- On considère la classe Employe qui est définie par :
  - deux variables d'instance cin et salaireBrutMensuel,
  - une méthode calculerIGR qui retourne l'impôt général sur les revenus salariaux.
  - La méthode getSalaireNetMensuel retourne le salaire net mensuel.
- Supposant que la formule de calcul de l'IGR diffère d'un pays à l'autre.
  - Au Maroc, par exemple le calcul s'effectue selon les cas suivant :
    - Si le salaire annuel est inférieur à 40000, le taux de l'IGR est : 5%
    - Si le salaire annuel est supérieur à 40000 et inférieur à 120000, le taux de l'IGR est : 20%
    - Si le salaire annuel est supérieur à 120000 le taux de l'IGR est : 42%
  - En Algérie, le calcul s'effectue en utilisant un taux unique de 35%.

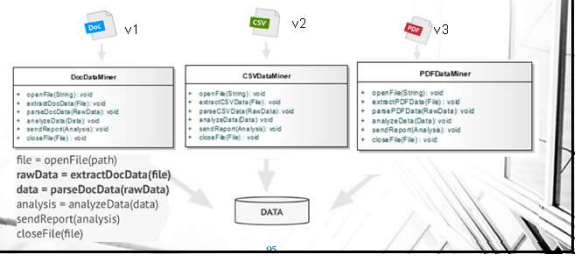
## Patrons de conception
### Patron Stratégie

- Exercice suite
  - Comme cette classe est destinée à être utilisée dans différent type de pays inconnus au moment du développement de cette classe,
  1. Identifier les méthodes qui vont subir des changements chez le client.
  2. En appliquant le pattern strategie, essayer de rendre cette classe fermée à la modification et ouverte à l'extension.
  3. Créer une application de test.
  4. Proposer une solution pour choisir dynamiquement l'implémentation de calcul de l'IGR.

94

## Patrons de conception
### Patrons de comportement

- Problème
  - Nous avons une application de data mining qui analyse les documents. Les utilisateurs alimentent l'application avec des documents de différents formats (PDF, DOC, CSV), et ils essaient d'extraire des données significatives dans un format uniforme.




95

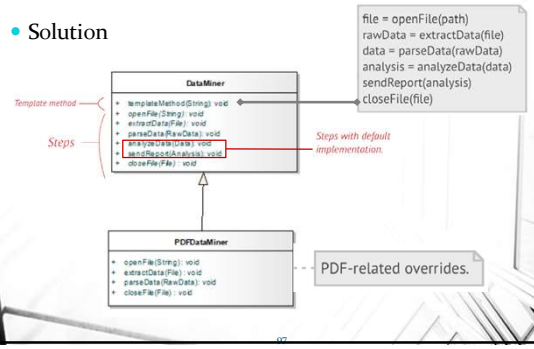
## Patrons de conception
### Patrons de comportement

- Quels sont les problèmes de cette application ?
  1. **Duplication du code :**
     1. les trois classes présentent du code similaire. Alors que le code pour gérer différents formats de données était entièrement différent dans toutes les classes. le code pour le traitement et l'analyse des données est presque identique.
  2. **Couplage fort avec la classe client :**
     - Le code client qui utilisait ces classes utilise des expressions conditionnels pour sélectionner la bonne méthode en fonction de la classe du traitement.
     - Si les trois classes de traitement avaient **une interface commune** ou une classe de base abstraite, vous seriez en mesure d'éliminer les conditions dans le code client et utiliser le polymorphisme lors de l'appel de méthodes sur un objet spécifique.

96

## Patrons de conception
Patrons de comportement

- Solution



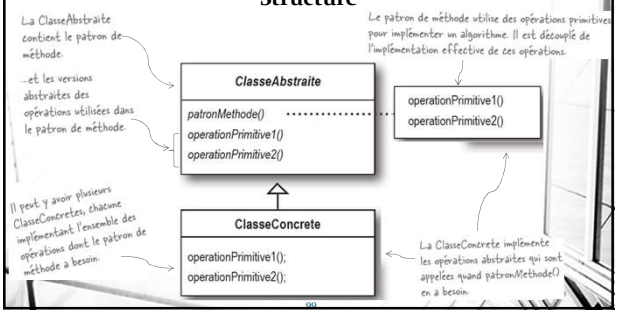

## Patrons de conception
Patron de méthode (Template Method)

**Définition**

- Définit le **squelette** d'un algorithme dans une opération en **reportant** certaines étapes à des **sous-classes**.
- Permet d'affiner certaines étapes d'un algorithme (dans les sous-classes) sans changer sa structure.

## Patrons de conception
Patron de méthode (Template Method)

**Structure**



## Patrons de conception
### Patron de méthode (Template Method)

**Participants**

- Les participants au pattern sont les suivants :
  - La classe abstraite **AbstractClass** (Order*Printer*) introduit la méthode « template method » ainsi que la signature des méthodes abstraites que cette méthode invoque.
  - La sous-classe concrète ***ConcretClass*** (TextOrderPrinter) implante les méthodes abstraites utilisées par la méthode « template method » de la classe abstraite. Il peut y avoir plusieurs classes concrètes.



## Patrons de conception
### Patron de méthode (Template Method)

Domaine d'application

- Le pattern est utilisé dans les cas suivants :
- Une classe partage avec une autre ou plusieurs autres classes du code identique qui peut être factorisé après que le ou les parties spécifiques à chaque classe aient été déplacées dans de nouvelles méthodes.
- Un algorithme possède une partie **invariable** et des parties **spécifiques** à différents types d'objets.



## Patrons de conception
### Patrons de comportement

- Exercice
  - Développer une application qui génère des factures en format text et en format HTML, l'application utilise une classe vente définie par un id, une date, et une liste d'items(String, Double)
  - L'algorithme :
    1. String start();
    2. String formatOrderNumber(Commande order);
    3. String formatOrderItems(Commande order);
    4. String formatTotal(Commande order) ;
    5. String end()
  - Indication :
    - utiliser la classe java PrintWriter(String filename)

## Patrons de conception
Patrons de comportement

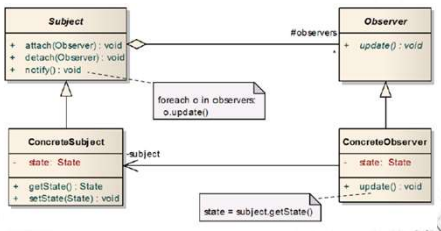
- Problème
  - *Nous voulons mettre à jour l'affichage d'un catalogue de véhicules en temps réel*
  - *Chaque fois que les informations relatives à un véhicule sont modifiées, nous voulons mettre à jour l'affichage de celles-ci.*
  - *Il peut y avoir plusieurs affichages simultanés.*

## Patrons de conception
Patron observer

- **Objectifs du pattern observer**
  - Le but du Patron Observer est de construire une dépendance 1 à plusieurs entre un objet (sujet) et des objets (observateurs)
  - chaque modification du sujet, càd à le changement d'état du sujet, implique une notification automatique des objets observateurs.

## Patrons de conception
Patron observer

**Structure**



## Patrons de conception
Patron observer

### Participants

- **Sujet :** est la classe abstraite qui introduit l'association avec les observateurs ainsi que les méthodes pour ajouter ou retirer des observateurs.
- **Observateur** est l'interface à implanter pour recevoir des notifications (méthode *actualise*).
- **SujetConcret** est une classe d'implantation d'un sujet. Un sujet envoie une notification quand son état est modifié.
- **ObservateurConcret** est une classe d'implantation d'un observateur. Celui-ci maintient une référence vers son sujet et implante la méthode *actualise*. Elle demande à son sujet des informations faisant partie de son état lors des mises à jour par invocation de la méthode getEtat

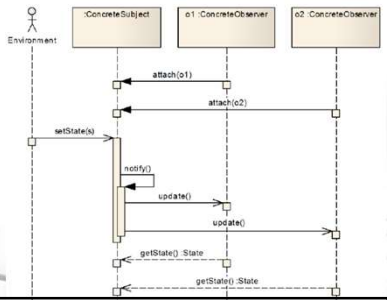
## Patrons de conception
Patron observer

### Actions

- **attach()**
  - Observateur « s'enregistre » au sujet
- **detach**()
  - Permet à un observateur de ne plus observer un sujet
- **notify**()
  - Appelée après que le sujet change d'état
- **update()**
  - Informe un observateur que de nouvelles données sont disponibles
- **getState()**
  - Recevoir l'état du sujet après une notification (méthode Pull)
  - Update avec un argument envoie les données aux observateurs (méthode Push)

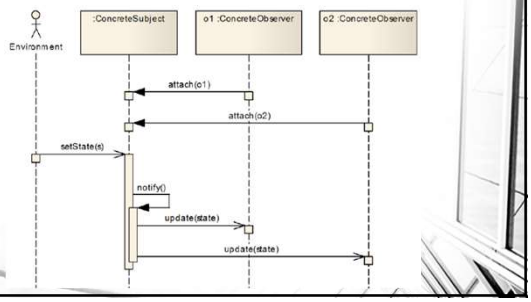
## Patrons de conception
Patron observer

### Interactions : méthode Pull



## Patrons de conception
Patron observer

**Interactions : méthode Push**



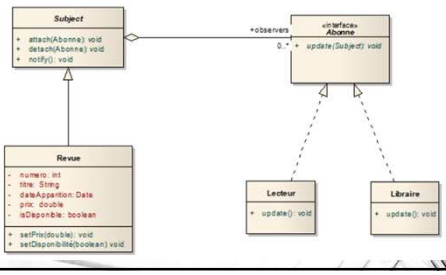

## Patrons de conception
Patron observer

**Domaine d'application**

- Le Patron est utilisé dans les cas suivants :
  - Une modification dans l'état d'un objet engendre des modifications dans d'autres objets qui sont déterminés dynamiquement.
  - Un objet veut prévenir d'autres objets sans devoir connaître leur type, c'est-à-dire sans être fortement couplé à ceux-ci.
  - On ne veut pas fusionner deux objets en un seul.

## Patrons de conception
Patron observer

Solution du problème



## Patrons de conception
Patron observer

Code source du problème

## Patrons de conception
Patron observer

Code

## Patrons de conception
Patron observer