



# DESIGN PATTERNS

## Analyse 3

2020-2021

## PLAN

### MCD

**Modèle conceptuel des données**

Diagramme de classes  
(rappels) Documentation

### MCT

**Modèle conceptuel des traitements**

Diagramme de Use Cases (UC)  
Documentation

Conceptuel

### UC Specification

Documentation de UC  
Interface utilisateur  
Diagramme d'activité (rappel)

### PTFE

**Plan de tests fonctionnels élémentaires**  
Documentation

Fonctionnel

### MTD-MTT

### UC Realization

Diagramme de séquence  
Diagramme de classes techniques

### Design Pattern

Technique

## Méthodes

## ● Réalité de terrain

- ◦ Designer un code réutilisable est difficile
  - Trouver les bons objets et abstractions
    - Est-ce que tout doit être un objet ?
    - Quelle interface doit avoir chaque objet ?
    - Héritage ou composition ?
    - ...

## ● Réalité de terrain

- - Designer un code réutilisable est difficile
    - Trouver la bonne flexibilité, modularité et élégance
  - Tout cela prend du temps pour émerger
    - Essais / Erreurs

- Réalité de terrain

- Mais de bons designs existent
  - Au fil du temps, on a collecté les bonnes idées et on en a extrait des “patterns”

## ● Définition

- ◦ Design patterns
  - décrivent une structure récurrente de design
    - extraient des noms et de l'abstraction de designs concrets
    - identifient des classes, des collaborations, des responsabilités
    - définissent le domaine d'application, les trade-offs, les conséquences et les problèmes liés aux langages
  - ont été découverts, ... pas inventés !

## ● Avantages

- ◦ Design patterns
  - montrent des solutions efficaces à des problèmes récurrents dans le développement
  - permettent d'améliorer la qualité du code produit
  - permettent la réutilisation d'architectures logicielles
  - permettent d'appliquer facilement les connaissances d'experts

## ● Utilité

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

*Christopher Alexander*



## ● Design Patterns

- ◦ 4 éléments :
  - **Le nom du pattern**
    - Définit un vocabulaire de design
  - **La description du problème**
    - Quand appliquer le pattern, dans quel contexte l'utiliser
  - **La description de la solution**
    - Générique !
    - Les éléments qui font le design (classes, ...) et leur relations, responsabilités et collaborations
  - **Conséquences**
    - Résultats et trade-offs de la solution

## ● Catalogue de design patterns

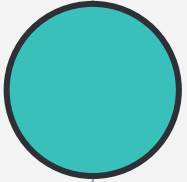
- - 3 types de design patterns :
    - de **création**
      - se focalisent sur le processus de création d'objets
    - de **structure**
      - se focalisent sur la composition de classes pour former de plus larges structures
    - de **comportement**
      - se focalisent sur les algorithmes et l'attribution des responsabilités entre les objets

## ● Catalogue de design patterns

- ◦ 2 niveaux (scope) :
  - de **classes**
    - se focalisent sur les relations entre les classes et leurs sous-classes
  - d'**objets**
    - se focalisent sur les relations entre objets qui peuvent changer durant l'exécution

## ● Catalogue de design patterns

- - **Singleton**
  - **Observer**
  - **Composite**
  - Proxy
  - Abstract Factory
  - Flyweight
  - Adapter
  - Chain of responsibility
  - Factory Method
  - Facade
  - Visitor
  - ...



# Singleton

*ou comment garder une seule instance pour une classe ?*

## ● Singleton

- - *Catégorie*
    - Création
  - *But*
    - Être sûr qu'une classe n'aura qu'une seule instance et fournir un point d'accès global.

## ● Singleton

- ◦ *Motivation*
  - Quand il ne peut y avoir qu'une instance.
  - Par exemple : beaucoup d'imprimantes mais seulement un Spooler d'impression.
  - Utiliser une variable globale contenant une seule instance ?
    - Vous ne pouvez pas être sûr qu'aucune autre instance ne sera créée.
  - La classe doit contrôler sa seule instance.

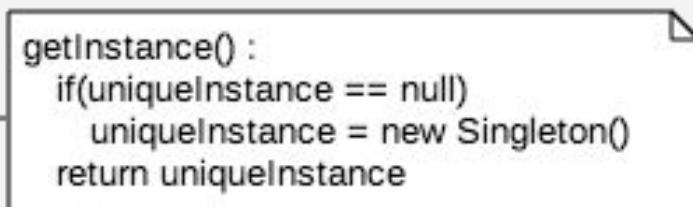
## ● Singleton

- ◦ *Application*
  - Utiliser le Singleton lorsque
    - Il ne doit exister qu'une seule instance d'une classe et elle doit être accessible par un point d'accès connu de tous
    - la seule instance doit pouvoir être extensible par sous-classe et les clients doivent être capables d'utiliser une instance étendue sans changer leur propre code.



## ● Singleton

### ● ◦ *Structure*



## ● Singleton

- ◦ *Participants & Collaborations*
  - Singleton
    - Définit une méthode d'instance qui laisse le client accéder à son unique instance. Instance est une méthode de classe qui soit retourne soit crée et retourne la seule instance.

## ● Singleton


- ◦ *Conséquences*
  - Contrôle d'accès à l'unique instance
    - Comme la classe Singleton encapsule sa seule instance, elle peut avoir un contrôle strict sur comment et quand les clients y accèdent.
  - Un namespace réduit
    - Le Singleton est une amélioration par rapport aux variables globales qui gardent des instances uniques.

## ● Singleton

- ◦ *Conséquences*
  - Permet le raffinement des opérations
    - Comme la classe Singleton peut être sous-classée, une application peut alors être configurée avec une instance d'une classe dont vous avez besoin à l'exécution.
  - Permet un nombre variable d'instances
    - La même approche peut être utilisée pour contrôler un nombre donné d'instances



## Singleton

- 
- *Conséquences*
    - Plus flexible que des opérations de classe

## Singleton

- *Exemple*

```
public class MazeFactory {  
    private static MazeFactory instance = null;  
    public static MazeFactory getInstance(){  
        if (instance == null){  
            instance = new MazeFactory();  
        }  
        return instance;  
    }  
    private MazeFactory();  
  
    // rest of the class  
    // ...  
}
```

## Singleton

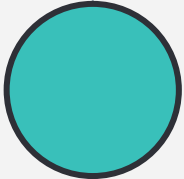
- *Exemple*

```
public class Main {  
  
    public static void main(String[] args) {  
  
        MazeGame gmg = new MazeGame();  
  
        //MazeFactory factory = new MazeFactory();  
        MazeFactory factory = MazeFactory.getInstance();  
  
        Maze mz = gmg.createMaze(factory);  
    }  
}
```

## ● Singleton

- ◦ *Utilisations connues*
  - A chaque fois que vous voulez une limite de création d'instances supplémentaires après la création de la première.
  - C'est utile pour limiter l'usage de la mémoire lorsque plusieurs instances ne sont pas nécessaires.





# Observer

*ou comment écouter les changements d'un objet en gardant un couplage faible ?*

## ● Observer

- - *Catégorie*
    - Comportement
  - *But*
    - Définir une relation de dépendance entre un et plusieurs objets telle que quand l'objet change d'état, tous ses dépendants sont notifiés et mis à jour automatiquement.

## ● Observer

- ◦ *Motivation*
  - Différents types d'éléments graphiques affichant les mêmes données d'une application.
  - Différents fenêtres montrant différentes vues sur le même modèle d'une application.

## ● Observer

- ◦ *Application*
  - Utiliser l'Observer lorsque
    - l'abstraction a deux aspects, un dépendant d'un autre. Encapsuler ces aspects dans des objets séparés vous permet de les modifier et les réutiliser de manière indépendante.

## ● Observer

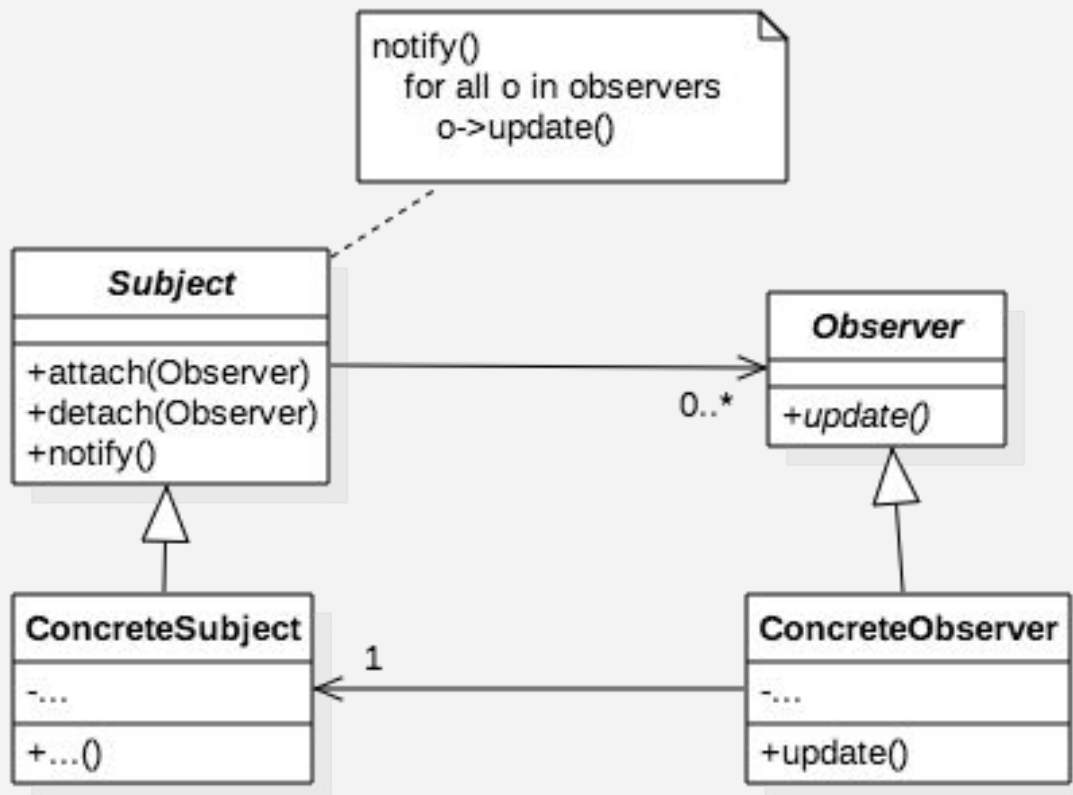
- ◦ *Application*
  - Utiliser l'Observer lorsque
    - un changement sur un objet demande d'en changer d'autres et que vous ne savez pas combien objets ont besoin d'être changé.

## ● Observer

- ◦ *Application*
  - Utiliser l'Observer lorsque
    - un objet doit notifier d'autres objets sans faire d'hypothèses sur quels sont ces objets. En d'autres mots, vous ne voulez pas que ces objets aient un couplage fort.

## Observer

- *Structure*



## ● Observer

- ◦ *Participants & Collaborations*
  - Subject
    - Connaît ses observateurs.
    - N'importe quel nombre d'observateurs peuvent observer un sujet.
    - Fournit une interface pour attacher et détacher les observateurs.



## ● Observer

- ◦ *Participants & Collaborations*
  - Observer
    - Définit l'interface de mise à jour des objets qui peuvent être notifiés des changements du sujet.

## ● Observer

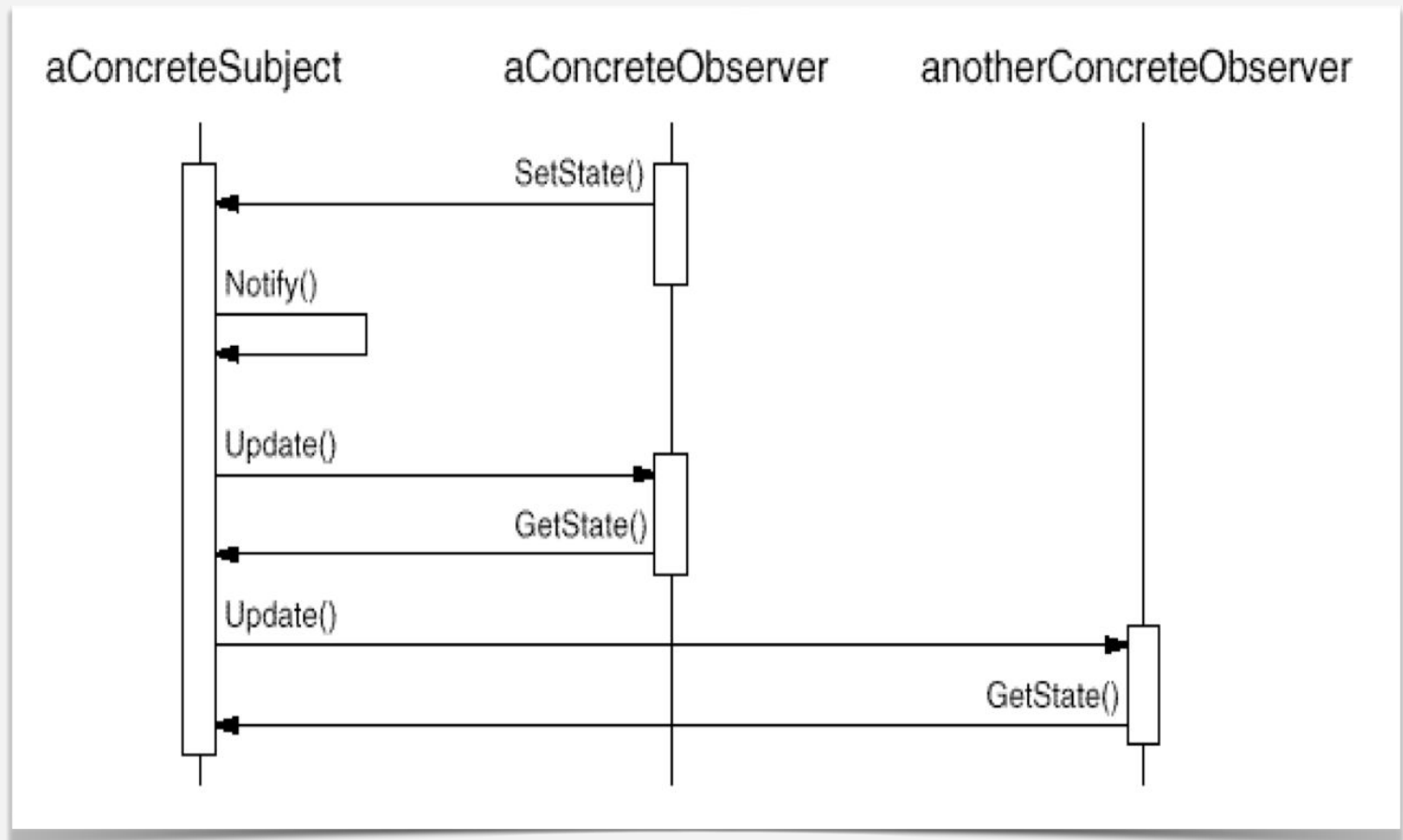
- ◦ *Participants & Collaborations*
  - ConcretSubject
    - Garde l'état qui intéresse les objets ConcreteObserver
    - Envoie une notification à ses observateurs lorsque son état change.

## ● Observer

- ◦ *Participants & Collaborations*
  - ConcretObserver
    - Maintient une référence vers un ConcreteSubject
    - Garde l'état qui doit être consistant avec celui du sujet
    - Implémente l'interface de mise à jour d'Observer

## Observer

- *Participants & Collaborations*



## ● Observer

- ◦ *Conséquences*
  - Abstraction et couplage faible entre Subject et Observer
    - Le sujet ne doit pas connaître la classe concrète de ses observateurs. Un sujet concret et un observateur concret peuvent être réutilisés indépendamment.

## ● Observer

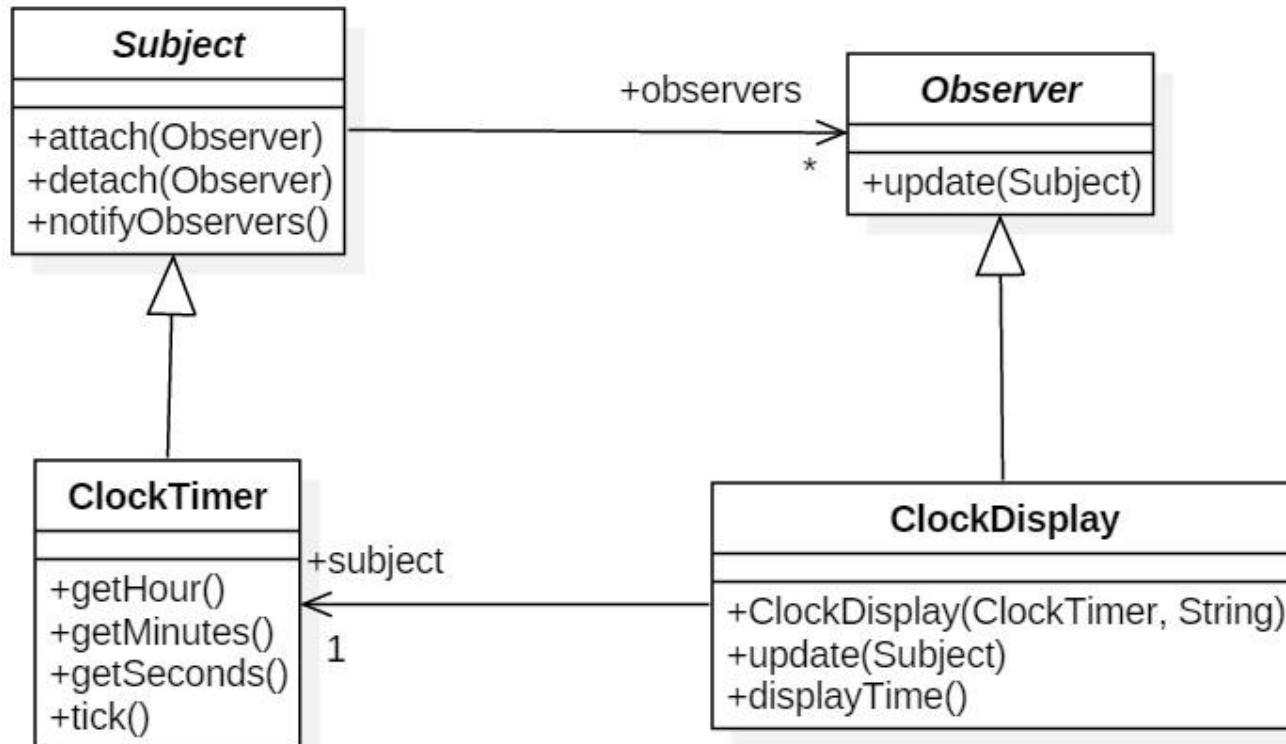
- ◦ *Conséquences*
  - Support pour une communication “broadcast”
    - La notification qu’envoie un sujet n’a pas besoin de spécifier un receveur, il sera envoyé à toutes les parties intéressées.

## ● Observer

- ◦ *Conséquences*
  - Mises à jour inattendues
    - Les observeurs ne sont pas conscients de la présence des autres observateurs. Une petite opération peut entraîner une cascade de mises à jour.

## Observer

### ◦ *Exemple*





## Observer

- *Exemple*

```
public abstract class Subject {  
    private List observers = new LinkedList();  
    protected Subject();  
    void attach(Observer o){  
        observers.add(o);  
    }  
    void detach(Observer o){  
        observers.remove(o);  
    }  
    void notifyObservers(){  
        ListIterator i = observers.listIterator(0);  
        while(i.hasNext()){  
            ((Observer) i.next()).update(this);  
        }  
    }  
}
```

## Observer

```
public abstract class Observer {  
    abstract void update(Subject changedSubject);  
    protected Observer();  
}
```

```
public class ClockTimer extends Subject {  
    int hour = 0;  
    int minutes = 0;  
    int seconds = 0;  
    int getHour(){  
        return hour;  
    }  
    int getMinutes(){  
        return minutes;  
    }  
    int getSeconds(){  
        return seconds;  
    }  
    //...
```



## Observer

```
void tick(){
    //updating the time
    if(seconds == 59){
        if (minutes == 59){
            if (hour == 23){
                hour = 0;
                minutes = 0;
                seconds = 0;
            }
            else {
                seconds = 0;
                minutes = 0;
                hour = hour + 1;
            }
        }
        else {
            seconds = 0;
            minutes = minutes + 1;
        }
    }
    else seconds = seconds + 1;
    notifyObservers();
}
```

## Observer

```
public class ClockDisplay extends Observer{
    private ClockTimer subject;
    private String clocktype;

    ClockDisplay(ClockTimer c, String type){
        subject = c;
        subject.attach(this);
        clocktype = type;
    }

    void update(Subject changedSubject){
        if (changedSubject == subject){
            displayTime();
        }
    }

    void displayTime(){
        System.out.println(clocktype + "-->" + subject.getHour()+
        ":" + subject.getMinutes()+ ":" + subject.getSeconds());
    }
}
```

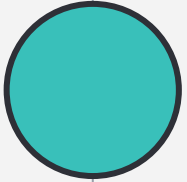
## Observer

```
//if not interrupted, will continue for 24 hours
public class Main {
    public static void main(String[] args) {
        ClockTimer c = new ClockTimer();
        ClockDisplay d1 = new ClockDisplay(c, "Digital");
        ClockDisplay d2 = new ClockDisplay(c, "Analog");
        int count = 0;
        while (count < (60*60*24)){
            c.tick();
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            count = count + 1;
        }
    }
}

Digital-->0:0:1
Analog-->0:0:1
Digital-->0:0:2
Analog-->0:0:2
```

## ● Observer

- - *Utilisations connues*
    - Utilisation dans le MVC
    - Dès que vous voulez dissocier des classes modèle d'autres choses.



# Composite

*ou comment gérer le tout et les parties de façon transparente ?*

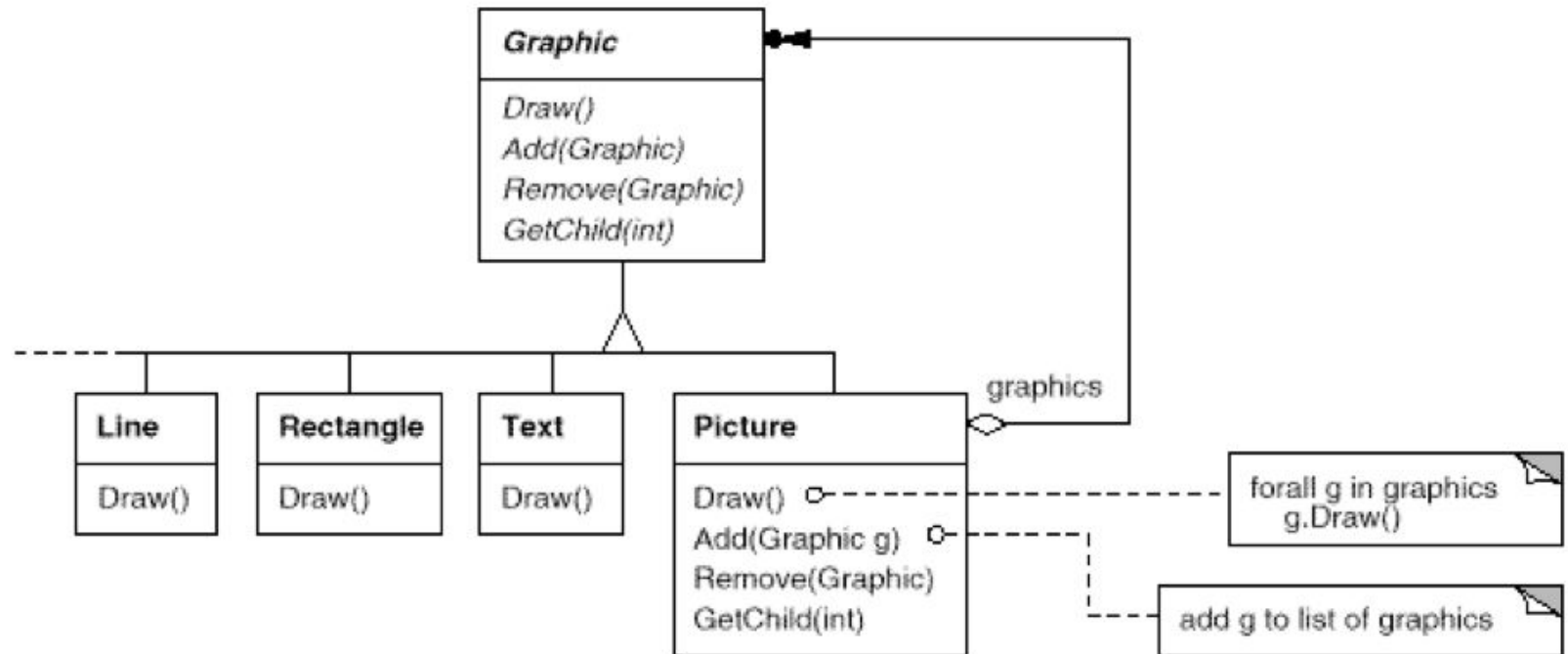
## ● Composite

- - *Catégorie*
    - Structure
  - *But*
    - Composer des objets dans une structure d'arbre pour représenter des relations tout-parties. Ce design pattern permet au client de traiter des objets individuels et les compositions d'objets de manière uniforme.



## Composite

- *Motivation*



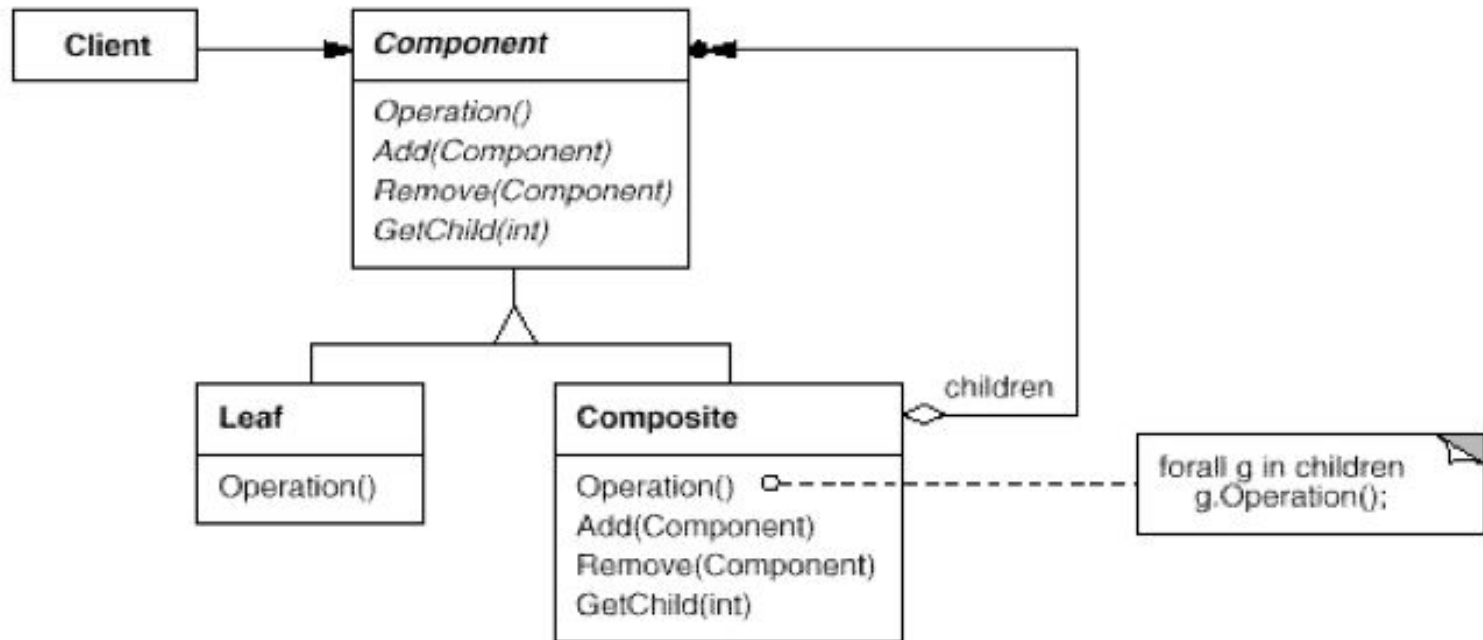
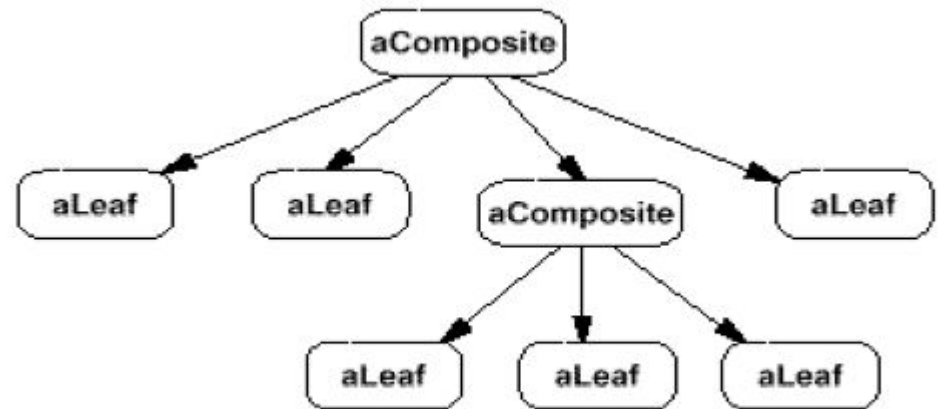
## ● Composite

### ● ◦ *Application*

- Utiliser le Composite lorsque
  - Vous voulez représenter des hiérarchies tout-parties
  - Vous voulez que les clients soient capable d'ignorer la différence entre des compositions d'objets et des objets individuels. Les clients vont traiter tous les objets dans la structure composite de manière uniforme.

## Composite

- *Structure*



## ● Composite

- ◦ *Participants & Collaborations*
  - Component
    - Déclare l'interface des objets dans la composition.
    - Implémente le comportement par défaut de l'interface commune à toutes les classes.
    - Déclare une interface pour accéder et gérer ses “enfants”.

## ● Composite


- ◦ *Participants & Collaborations*
  - Leaf
    - Représente les feuilles dans la composition. Une feuille n'a pas d'enfant.
    - Définit le comportement pour les objets primitifs de la composition.

## ● Composite

- ◦ *Participants & Collaborations*
  - Composite
    - Définit le comportement pour les composants ayant des enfants.
    - Stocke les enfants.
    - Implémente les opérations sur les enfants déclarées dans Component.



## Composite

- 
- *Participants & Collaborations*
    - Client
      - Manipule les objets de la composition à travers l'interface de Component.

## ● Composite

- ◦ *Conséquences*
  - Définit une hiérarchie de classes avec des objets primitifs et des objets composés.
  - Permet de rendre le client plus simple.
    - Il manipule de la manière uniforme les objets primitifs et composés.
  - Création simplifiée de nouveaux composants.
  - ! Peut rendre votre design trop général.



## Composite

### ◦ *Exemple*

```
abstract class Equipment {
    String name;
    public String name(){
        return name;
    }
    abstract int power();
    abstract int netPrice();
    abstract void add(Equipment e);
    abstract void remove(Equipment e);
    public Iterator createIterator(){
        return new NullIterator();
    }
    protected Equipment(String n){
        name = n;
    }
}
```

## Composite

```
abstract class CompositeEquipment extends Equipment {
    ArrayList l;
    public CompositeEquipment(String name){
        super(name);
        l = new ArrayList();
    }
    abstract int power();
    public int netPrice(){
        Iterator i = createIterator();
        int total = 0;
        while(i.hasNext()){
            Equipment e = (Equipment)i.next();
            total += e.netPrice();
        }
        return total;
    }
    public void add(Equipment e){
        l.add(e);
    }
    //...
```



## Composite

```
//...  
    public void remove(Equipment e){  
        l.remove(e);  
    }  
    public Iterator createIterator(){  
        return l.listIterator();  
    }  
}
```



## Composite

```
public class NullIterator implements Iterator {  
    public boolean hasNext(){  
        return false;  
    }  
    public Object next() throws NoSuchElementException{  
        throw new NoSuchElementException();  
    }  
    public void remove() throws IllegalStateException{  
        throw new IllegalStateException();  
    }  
}
```



## Composite

```
public class Cabinet extends CompositeEquipment {  
    public Cabinet(String name){  
        super(name);  
    }  
    public int power(){  
        return 0;  
    }  
    public int netPrice(){  
        int total = 60 + super.netPrice();  
        return total;  
    }  
}
```



## Composite

```
public class Chassis extends CompositeEquipment {  
    public Chassis(String name){  
        super(name);  
    }  
    public int power(){  
        return 0;  
    }  
    public int netPrice(){  
        int total = 40 + super.netPrice();  
        return total;  
    }  
}
```



## Composite

```
public class Bus extends CompositeEquipment {  
    public Bus (String name){  
        super(name);  
    }  
    public int power(){  
        return 40;  
    }  
    public int netPrice(){  
        int total = 100 + super.netPrice();  
        return total;  
    }  
}
```

## Composite

```
public class Card extends Equipment{  
    public Card(String name){  
        super(name);  
    }  
    public int power(){  
        return 60;  
    }  
    public int netPrice(){  
        return 100;  
    }  
    public void add(Equipment e){}  
    public void remove(Equipment e){}  
}
```



## Composite

```
public class Floppydisk extends Equipment {  
    public Floppydisk(String name){  
        super(name);  
    }  
    public int power(){  
        return 60;  
    }  
    public int netPrice(){  
        return 50;  
    }  
    public void add(Equipment e){}  
    public void remove(Equipment e){}  
}
```

## Composite

```
public class Main {  
    public static void main(String[] args) {  
        Cabinet cabinet = new Cabinet("PC Cabinet");  
        Chassis chassis = new Chassis("PC Chassis");  
        cabinet.add(chassis);  
        Bus bus = new Bus("MCA bus");  
        bus.add(new Card("NetworkCard"));  
        chassis.add(bus);  
        chassis.add(new Floppydisk("3.5 Floppy"));  
        System.out.println("The price is " +  
cabinet.netPrice());  
    }  
}
```

Console:

The price is 350

## ● Composite

- - *Utilisations connues*
    - Très répandu dans les systèmes OO