

Design Patterns

Christophe Damas

José Vander Meulen

Design Patterns

- Identifient des solutions récurrentes et bonnes à des problèmes récurrents en informatique.
 - Un développeur compétent se doit d'être capable d'identifier ces problèmes.
 - Un développeur compétent se doit d'être capable d'implémenter la solution correspondante.
 - Un développeur compétent se doit d'être capable de communiquer avec ses pairs en utilisant le vocabulaire des patterns.

Patterns

- Créationnel (instancier les objets)
 - Builder
 - Abstract Factory
 - Factory Method
 - Singleton
 - Prototype
- Structurel (composition des classes et objets)
 - Adapter
 - Facade
 - Composite
 - Decorator
 - Flyweight
- Comportemental (communication entre les objets)
 - Strategy
 - Command
 - Visitor
 - Observer
 - Template Method
 - Chain of responsibility
 - Iterator
 - State

Patterns

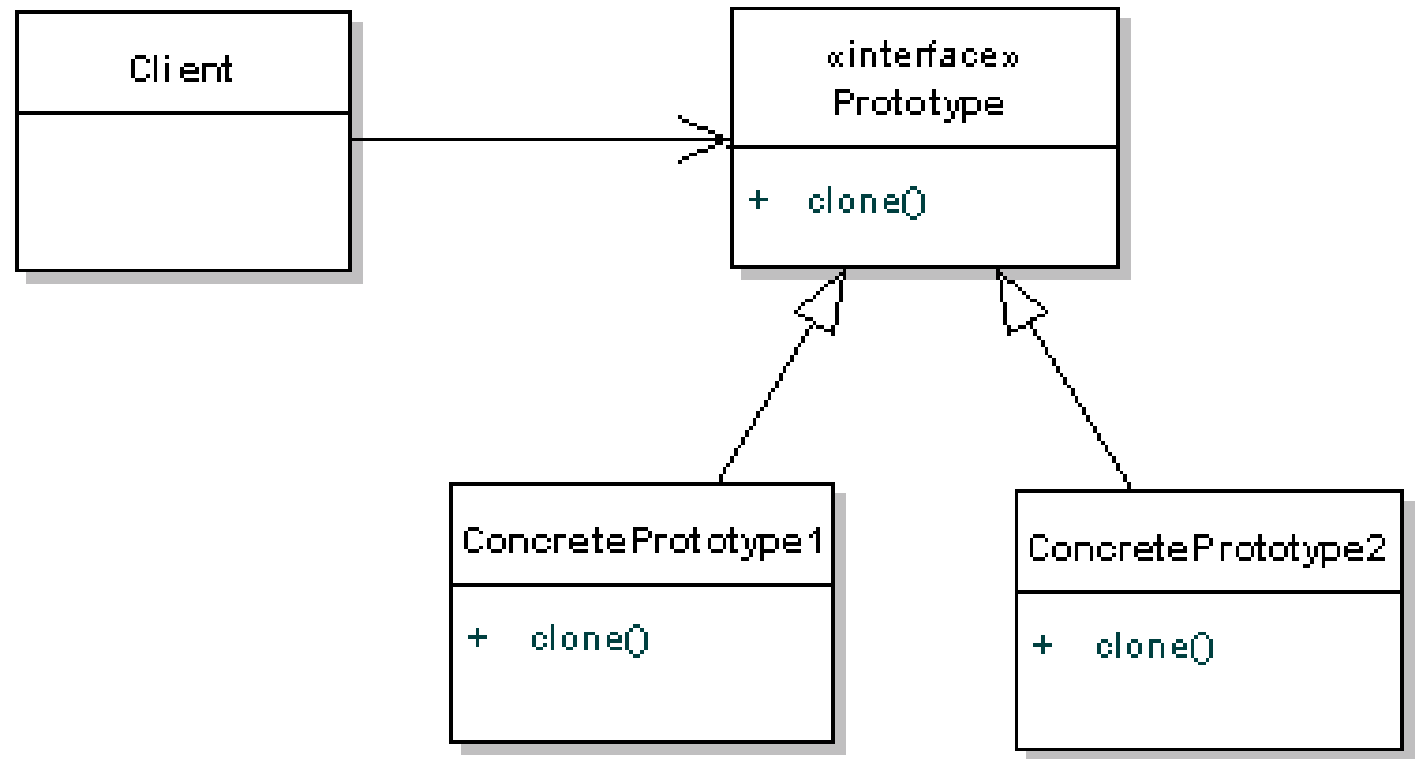
- Créationnel (instancier les objets)
 - Builder
 - Abstract Factory
 - Factory Method
 - **Singleton**
 - **Prototype**
- Structurel (composition des classes et objets)
 - **Adapter**
 - **Facade**
 - **Composite**
 - **Decorator**
 - **Flyweight**
- Comportemental (communication entre les objets)
 - **Strategy**
 - **Command**
 - **Visitor**
 - **Observer**
 - **Template Method**
 - Chain of responsibility
 - **Iterator**
 - State

Singleton

Singleton	
-	<u>singleton : Singleton</u>
-	Singleton()
+	<u>getInstance() : Singleton</u>

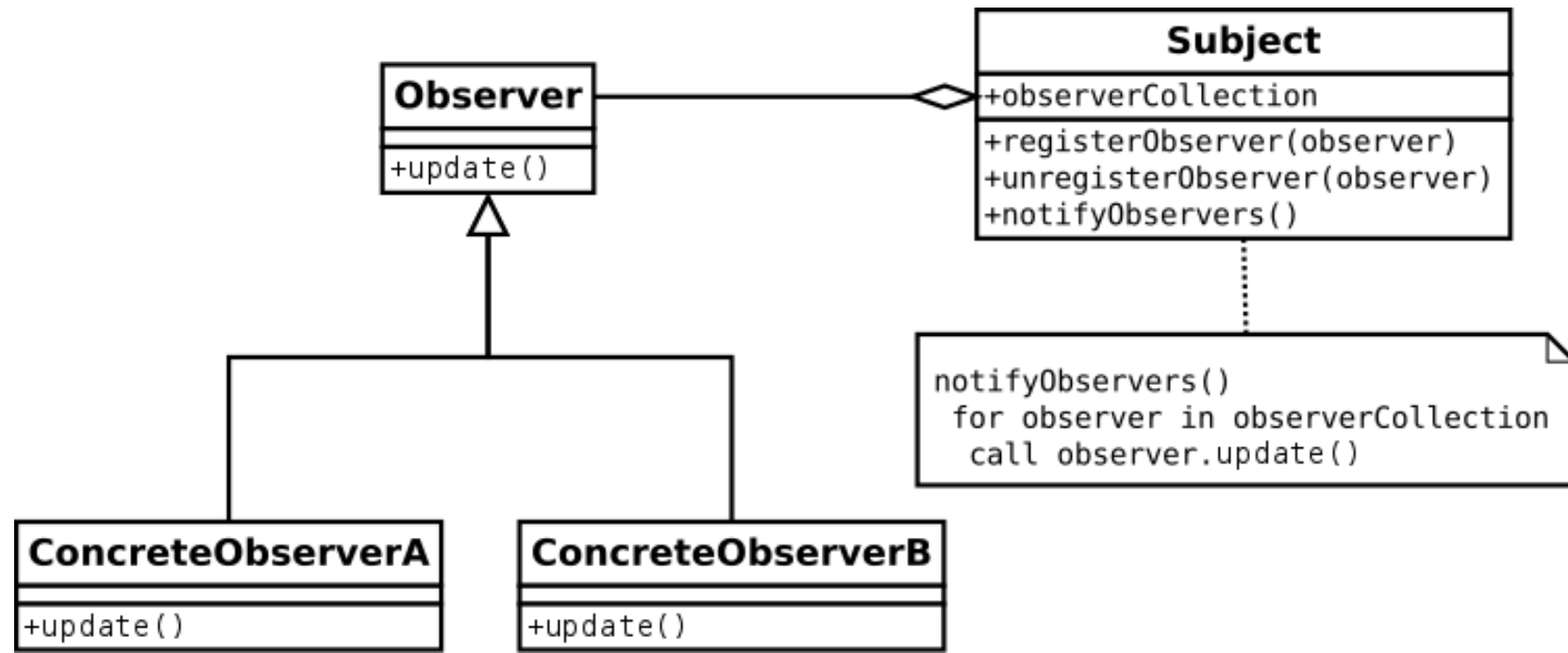
- Vous l'avez déjà vu en Atelier Java!
- Une seule instance pour une classe donnée.

Prototype



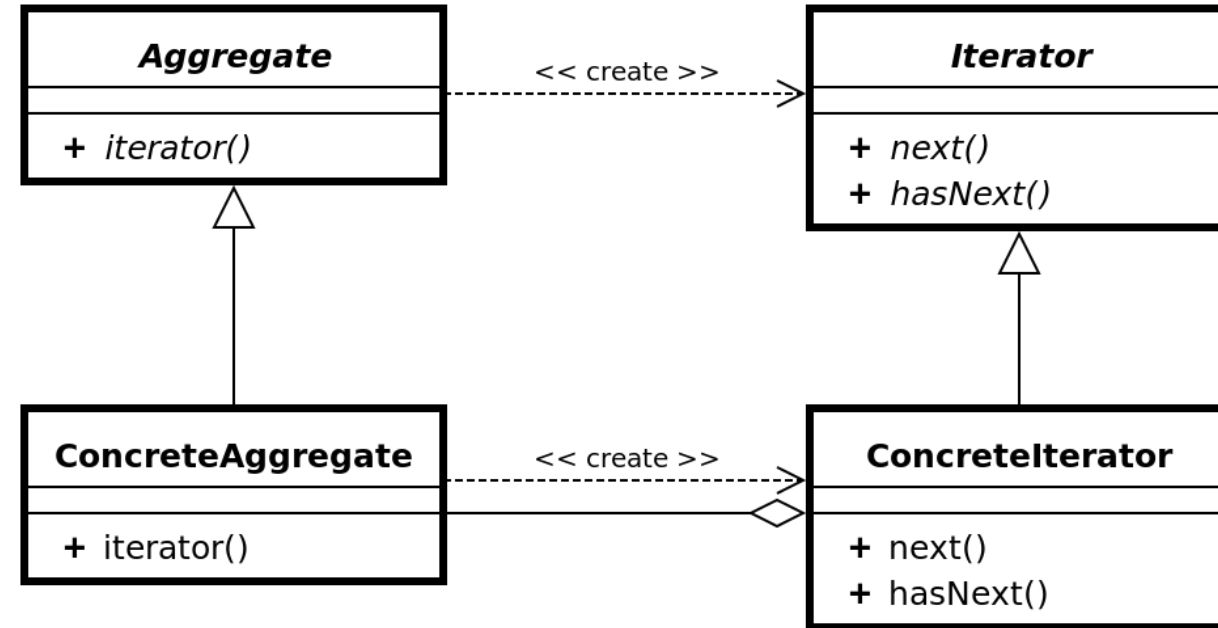
- Vous l'avez déjà rencontré en Ateliers Java!
- La création d'objets est basé sur le clonage d'un objet passé en paramètre.

Observer



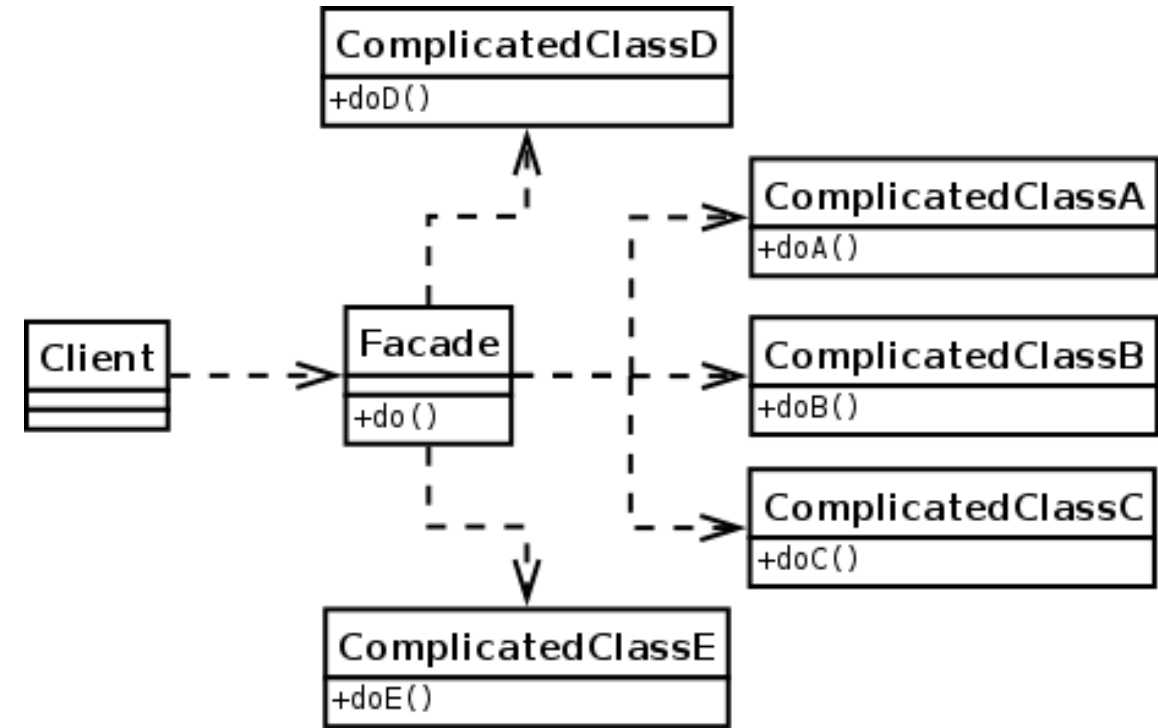
- Vous l'avez déjà rencontré!
- Un objet observé (sujet) retient une liste d'observateurs et les notifie de changement d'état en appelant une de leurs méthodes

Iterator



- Vous l'avez vu en APOO
- Fournit un moyen d'accéder séquentiellement aux éléments d'une collection d'objet sans connaître sa représentation physique

Facade



- Vous l'avez déjà vu en projet AE avec l'UCC
- La facade présente une signature simplifiée sur un ensemble complexe d'objets.
- Rend le système plus facile à utiliser

Flyweight: exemple introductif

- Qu'affiche ce code ?

```
public class FlyweightTest {  
    public static void main(String[] args) {  
        Integer i=1; Integer j=1;  
        System.out.println(i==j);  
    }  
}
```

- Est-ce que ce code compile ?

Flyweight: exemple introductif

- Java va en fait exécuter le programme suivant:

```
public class FlyweightTest {  
    public static void main(String[] args) {  
        Integer i=Integer.valueOf(1);  
        Integer j=Integer.valueOf(1);  
        System.out.println(i==j);}  
}
```

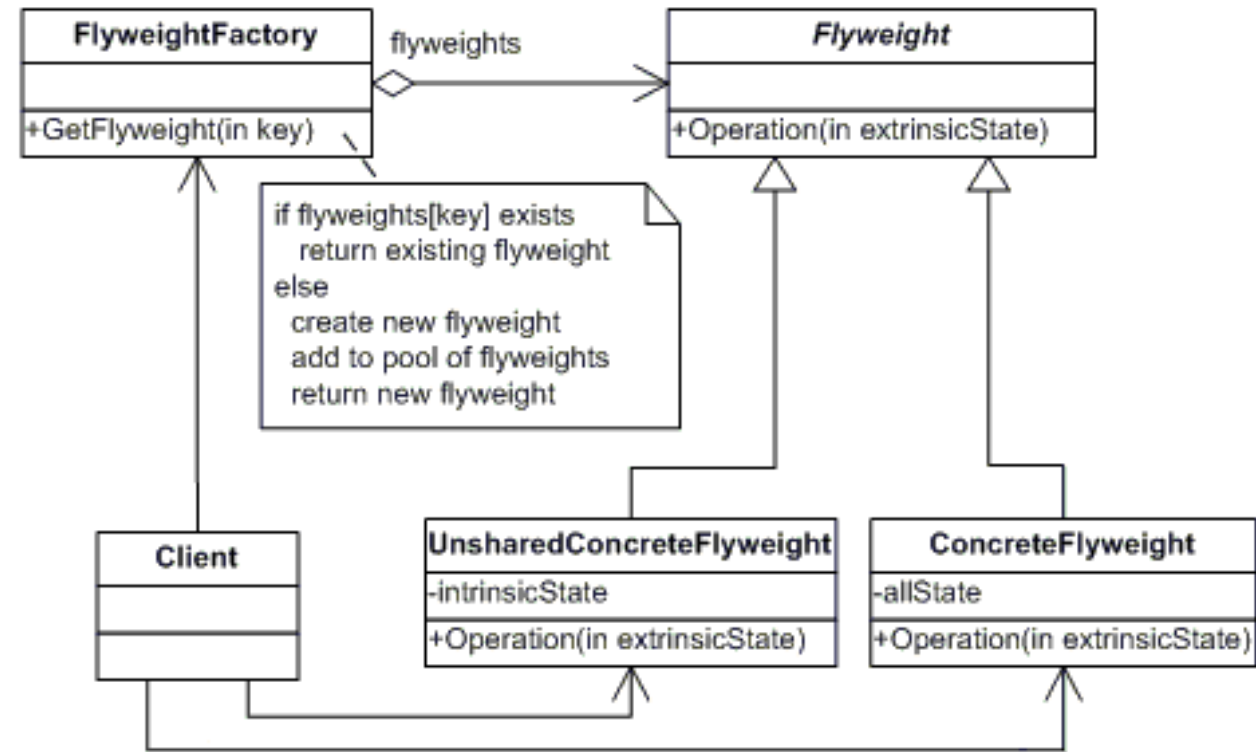
- Qu'affiche ce programme ? Comment savoir ?

Flyweight: exemple introductif

```
public class Integer{
    ...
    public static Integer valueOf(int i) {
        final int offset = 128;
        if (i >= -128 && i <= 127) { // must cache
            return IntegerCache.cache[i + offset];}
        return new Integer(i);}

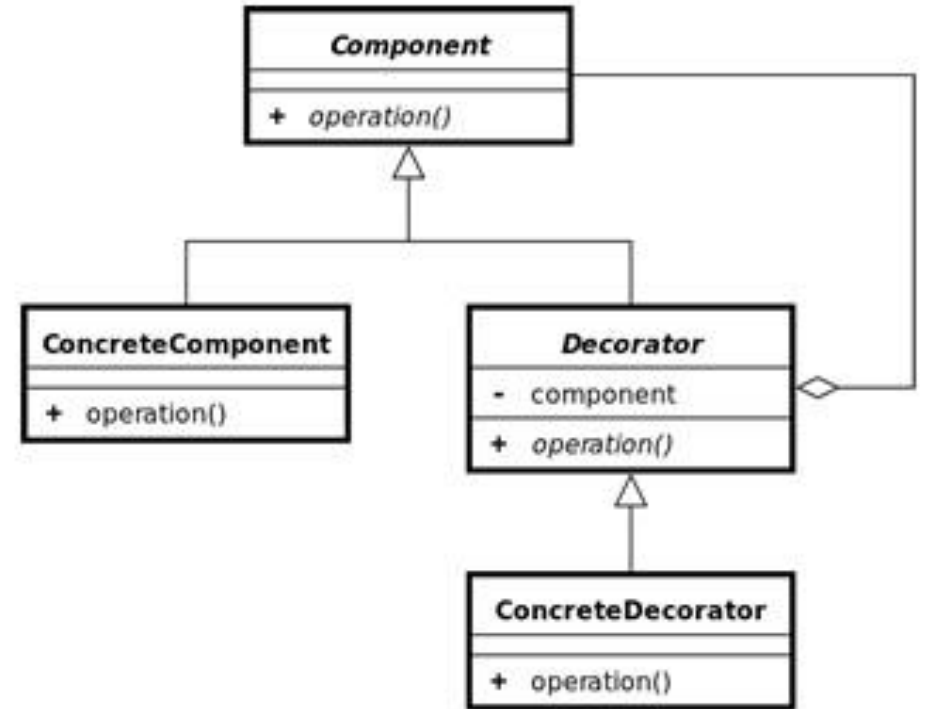
    private static class IntegerCache {
        private IntegerCache() {}
        static final Integer cache[] = new Integer[-(-128) + 127 + 1];
        static {
            for (int i = 0; i < cache.length; i++)
                cache[i] = new Integer(i - 128);}
    }
}
```

Flyweight



- Réduit le coût de création/manipulation de nombreux objets similaires.
- A la place d'instancier plusieurs objets identiques, on n'en instancie qu'un que l'on réutilise
- La classe Integer utilise le pattern Flyweight

Decorator

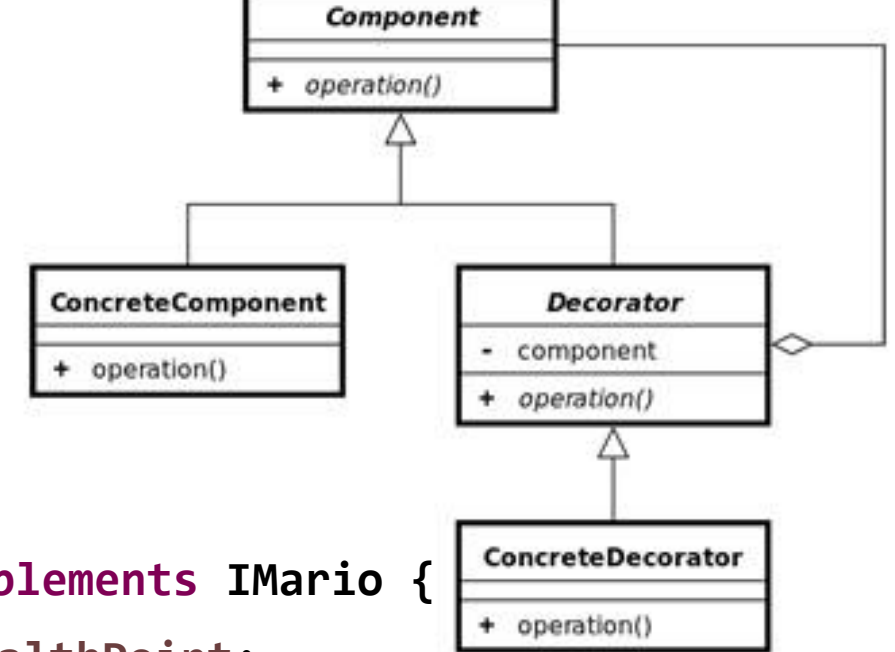


- Permet d'augmenter le comportement d'un objet dynamiquement
- Exemple: Mario

Decorator: example

```
public interface IMario {  
    void jump();  
    void moveForward();  
    void takeDamage();  
    void update();  
}
```

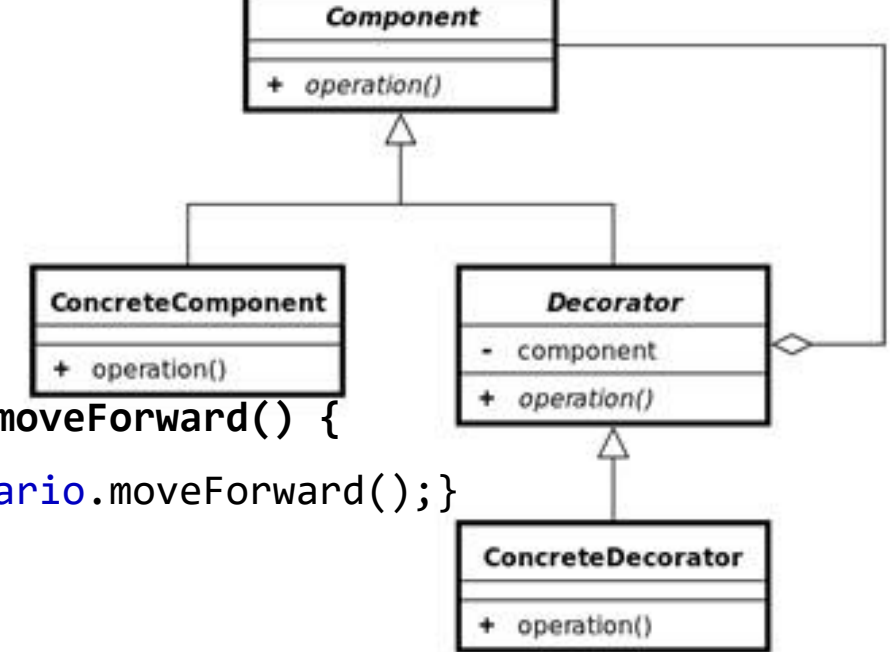
```
public class Mario implements IMario {  
    private int healthPoint;  
    public Mario(int healthPoint) {  
        super();  
        this.healthPoint = healthPoint;  
    }  
    public void jump() { // ... }  
    public void moveForward() { // ... }  
    public void takeDamage() {  
        healthPoint--;  
    }  
    public void update() { // ... }  
}
```



Decorator: Mario

```
public class StarMario implements IMario {  
    private IMario decoratedMario;  
    private int timer = 1000;  
  
    public StarMario(IMario decoratedMario) {  
        this.decoratedMario = decoratedMario;  
    }  
  
    @Override  
    public void takeDamage() {  
        // StarMario does not take damage  
    }  
  
    @Override  
    public void jump() {  
        decoratedMario.jump();  
    }  
}
```

```
@Override  
public void moveForward() {  
    decoratedMario.moveForward();  
}  
  
@Override  
public void update() {  
    timer--;  
    if (timer == 0) {  
        removeStar();  
        decoratedMario.update();  
    }  
  
    public void removeStar() {  
        GameSingleton.instance.mario =  
                                                    decoratedMario  
    }  
}
```



Template Method / Strategy

- Même objectif: Définition d'une famille d'algorithmes
- Exemple: trier une liste de personnes selon différents critères

```
public class Personne {  
    private String nom;  
    private int age;  
    private int tailleEnCm;  
    private int poids;  
    public Personne(String nom, int age, int tailleEnCm, int poids) {  
        super();  
        this.nom = nom;  
        this.age = age;  
        this.tailleEnCm = tailleEnCm;  
        this.poids = poids;}  
}
```

Template Method / Strategy: exemple

```
public class ListePersonnesTrieParAge {  
    private Personne[] personnes;  
    public ListePersonnesTrieParAge(Personne... personnes) {  
        this.personnes = personnes;  
    }  
    public void tri() {  
        int cpt;  
        Personne element;  
        for (int i = 1; i < personnes.length; i++) {  
            element = personnes[i];  
            cpt = i - 1;  
            while (cpt >= 0 && personnes[cpt].getAge() > element.getAge()) {  
                personnes[cpt + 1] = personnes[cpt];  
                cpt--;}  
            personnes[cpt + 1] = element;}  
        for (Personne pers : personnes)  
            System.out.println(pers);}  
    }  
}
```

```
public class ListePersonnesTrieParNom {  
    private Personne[] personnes;  
    public ListePersonnesTrieParAge(Personne... personnes) {  
        this.personnes = personnes;  
    }  
    public void tri() {  
        int cpt;  
        Personne element;  
        for (int i = 1; i < personnes.length; i++) {  
            element = personnes[i];  
            cpt = i - 1;  
            while (cpt >= 0 && personnes[cpt].getNom() > element.getNom()) {  
                personnes[cpt + 1] = personnes[cpt];  
                cpt--;}  
            personnes[cpt + 1] = element;}  
        for (Personne pers : personnes)  
            System.out.println(pers);}  
    }  
}
```

Exemple refactoré avec Strategy

```
public class ListePersonnes {
    private Personne[] personnes;

    public ListePersonnes(Personne... personnes) {
        this.personnes = personnes;
    }

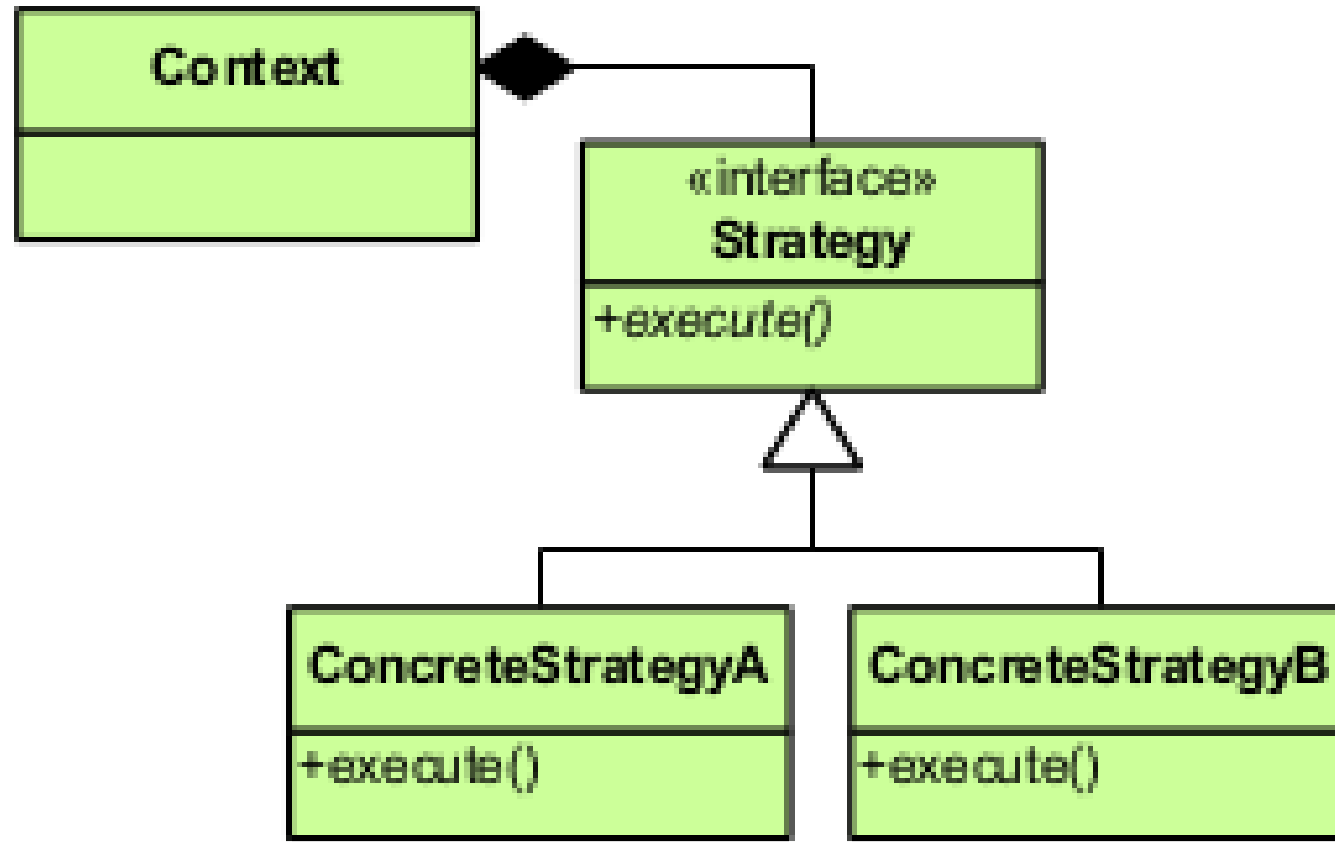
    public void tri(ComparatorStrategy cs) {
        int cpt; Personne element;
        for (int i = 1; i < personnes.length; i++) {
            element = personnes[i]; cpt = i - 1;
            while (cpt >= 0 &&
                cs.compare(personnes[cpt], element)) {
                personnes[cpt + 1] = personnes[cpt]; cpt--;
                personnes[cpt + 1] = element;
            }
            for (Personne pers : personnes)
                System.out.println(pers);
        }
    }
}
```

```
public interface ComparatorStrategy{
    public boolean compare(Personne p1, Personne p2);
}

public class NomStrategy implements ComparatorStrategy{
    @Override
    public boolean compare(Personne p1, Personne p2) {
        return (p1.getNom().compareTo(p2.getNom())>0);
    }
}

public class Main{
    public static void main(String[] args) {
        Personne cd = new Personne("Damas", 40, 184, 70);
        Personne jvm = new Personne("Vander Meulen", 39, 182, 72);
        Personne grolaux = new Personne("Grolaux", 45, 180, 74);
        ListePersonnes lp = new ListePersonnes(cd, jvm, grolaux);
        lp.tri(new NomStrategy());
    }
}
```

Strategy

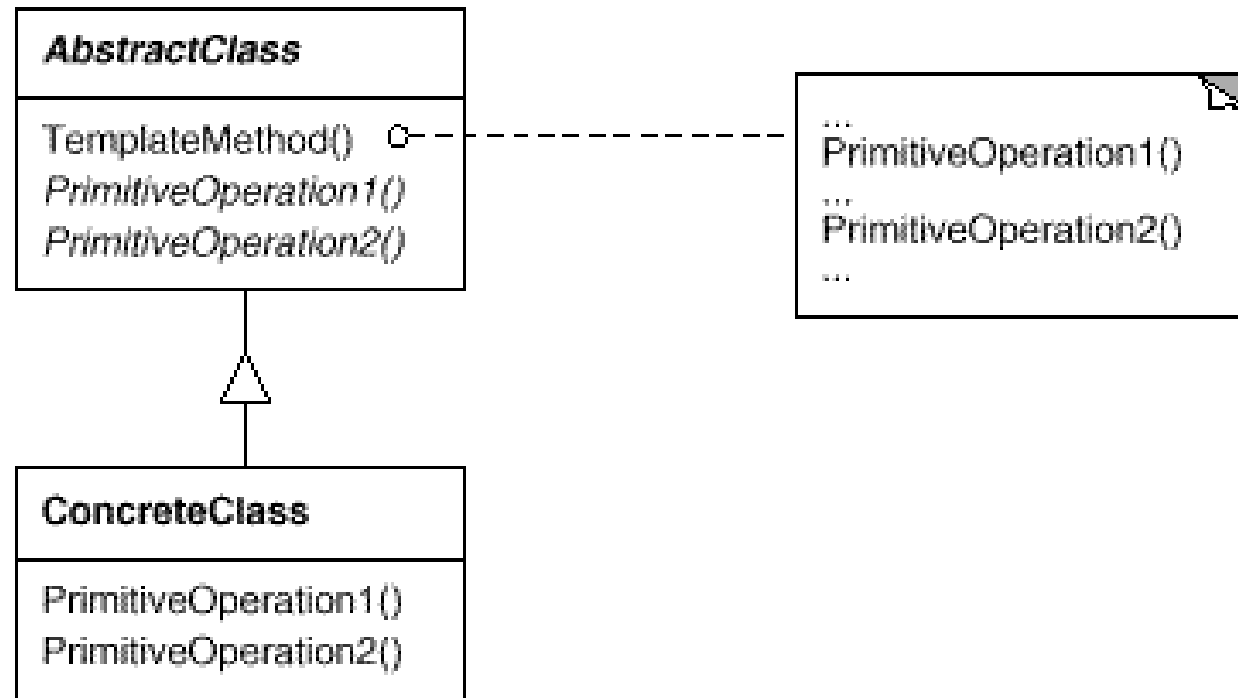


Exemple refactoré avec Template Method

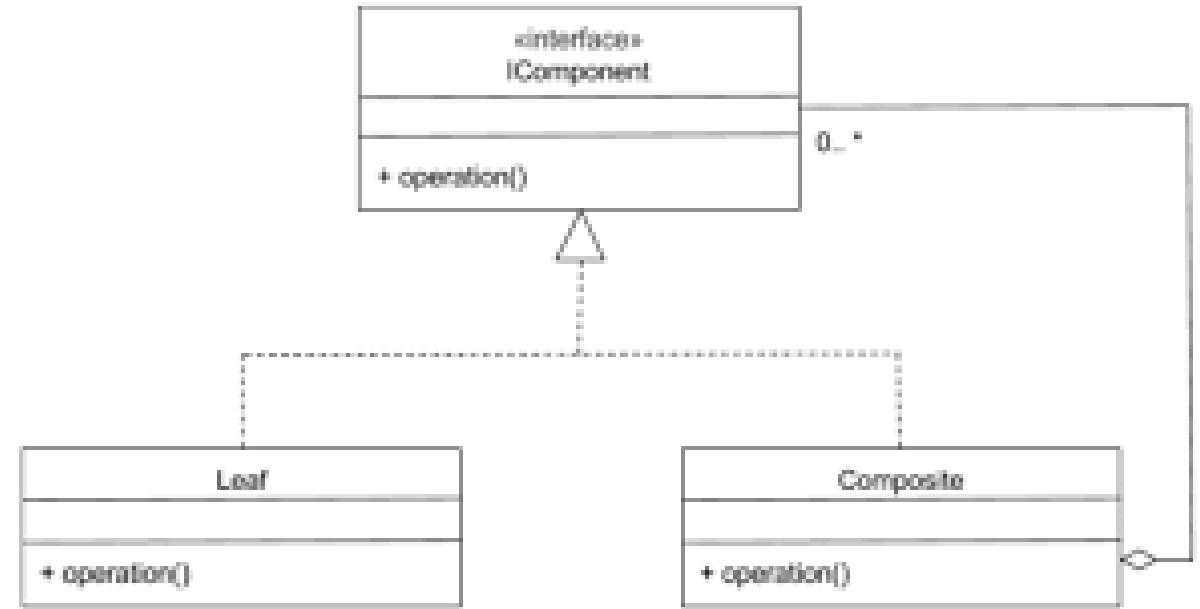
```
public abstract class ListePersonnes {  
    private Personne[] personnes;  
    public ListePersonnes(Personne... personnes) {  
        this.personnes = personnes;  
    }  
    public void tri() {  
        int cpt; Personne element;  
        for (int i = 1; i < personnes.length; i++) {  
            element = personnes[i]; cpt = i - 1;  
            while (cpt >= 0 &&  
                this.compare(personnes[cpt], element)) {  
                personnes[cpt + 1] = personnes[cpt]; cpt--;}  
            personnes[cpt + 1] = element;}  
        for (Personne pers : personnes)  
            System.out.println(pers);}  
  
    public abstract boolean compare(Personne personne,  
    Personne element);  
}
```

```
public class ListePersTrieParAge extends ListePersonnes{  
    public ListePersTrieParAge(Personne... personnes) {  
        super(personnes);}  
    @Override  
    public boolean compare(Personne p1, Personne p2) {  
        return p1.getAge() > p2.getAge();}  
}  
  
public class Main{  
    public static void main(String[] args) {  
        Personne cd = new Personne("Damas", 40, 184, 70);  
        Personne jvm = new Personne("Vander Meulen", 39, 182, 72);  
        Personne grolaux = new Personne("Grolaux", 45, 180, 74);  
        ListePersonnes lp = new  
            ListePersTrieParAge(cd, jvm, grolaux);  
        lp.tri();  
    }  
}
```

Template Method



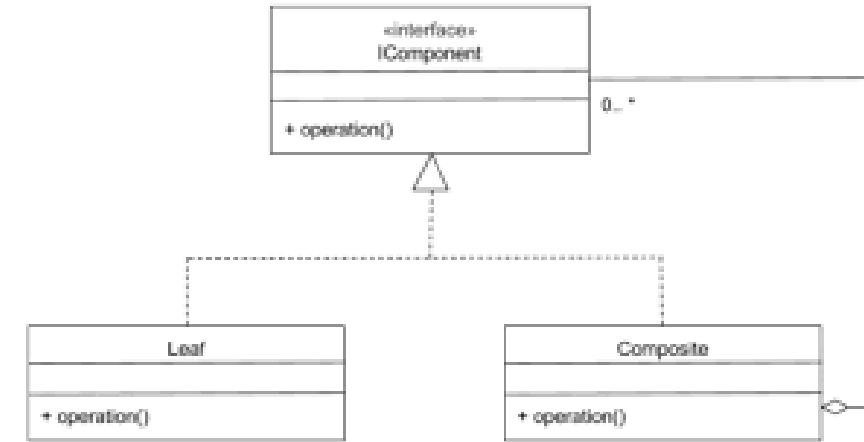
Composite



- Vous l'avez déjà vu: les arbres.
- Combine plusieurs objets similaires pour se comporter comme un objet unique.
- Exemple: logiciel de dessin

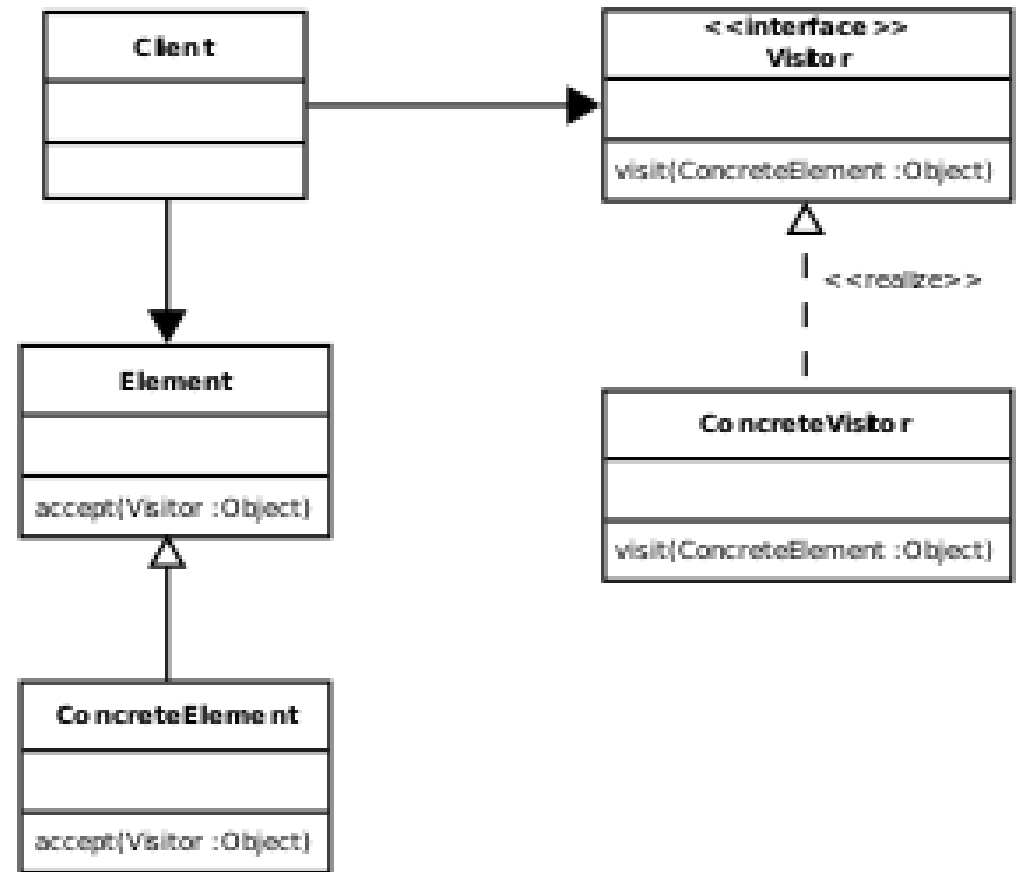
Composite: example

```
public interface Shape {  
    void move(int x, int y);  
    void draw();  
}  
  
public class Circle implements Shape {  
    private double diameter;  
    public Circle(double diameter) {  
        this.diameter=diameter;}  
    @Override  
    public void move(int x, int y) {  
        // move a Circle}  
    @Override  
    public void draw() { // draw a Circle}  
}  
  
public class Square implements Shape {...}
```



```
public class CompoundShape implements Shape {  
    private List<Shape> shapes= new ArrayList<Shape>();  
    @Override  
    public void move(int x, int y) {  
        for(Shape s: shapes) {s.move(x, y);}  
    }  
    @Override  
    public void draw() {  
        for(Shape s: shapes) {s.draw();}  
    }  
  
    public void addShape(Shape s) {  
        shapes.add(s);  
    }  
}
```


Visitor



- Vous l'avez déjà vu en SD1: Baladeur
- Représente une opération à appliquer sur les éléments d'une structure d'objets (souvent un composite)
 - Ressemble à une Strategy avec autant de méthodes que de type de nœuds différents.
- Exemple: Exporter les différentes formes dans un fichier xml

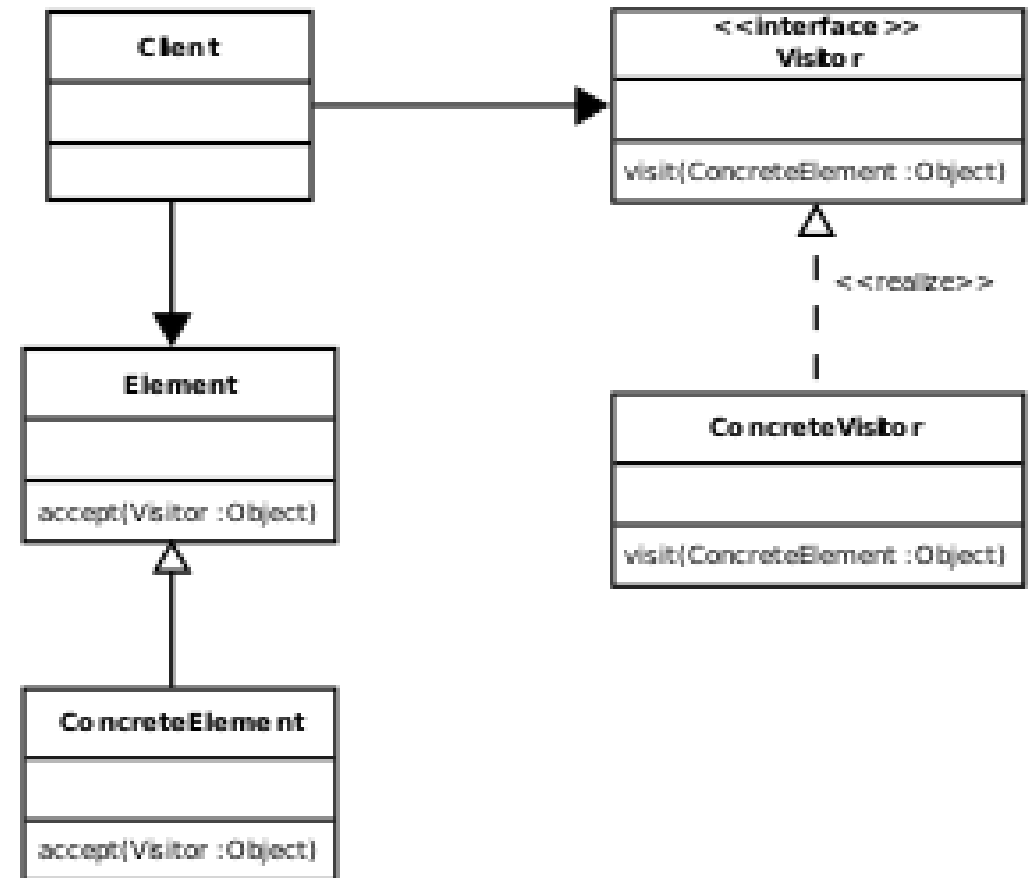
Visitor: exemple

```
public interface Visitor {  
    String visitCircle(Circle circle);  
    String visitSquare(Square s);  
    String visitCompoundShape(CompoundShape cs);  
}
```

```
public interface Shape {  
    void move(int x, int y);  
    void draw();  
    String accept(Visitor v);  
}
```

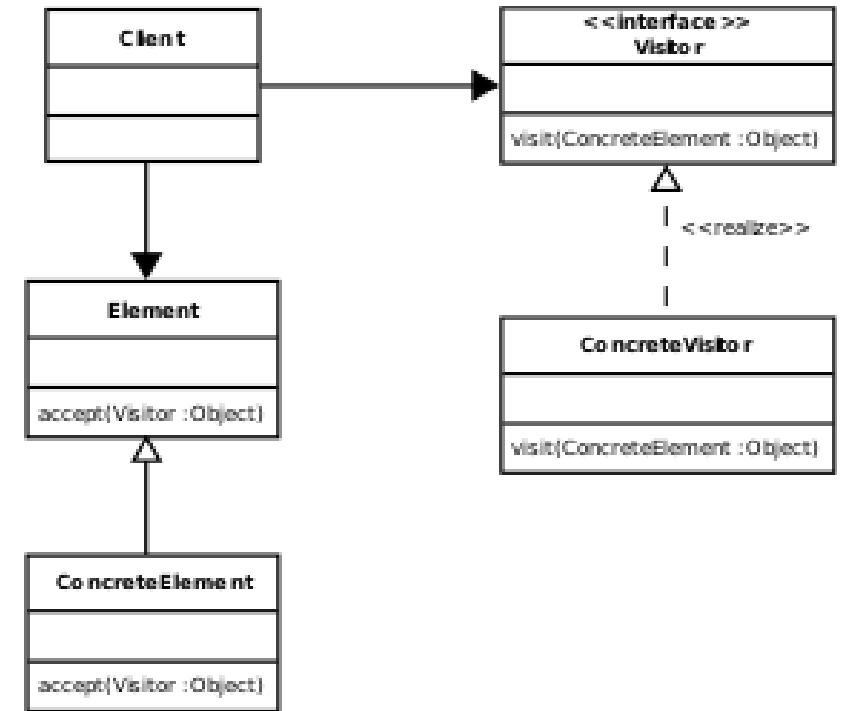
```
public class Circle implements Shape {  
    ...  
    @Override  
    public String accept(Visitor v) {  
        return v.visitCircle(this);  
    }  
}
```

```
public class CompoundShape implements Shape {  
    ...  
    @Override  
    public String accept(Visitor v) {  
        return v.visitCompoundShape(this);  
    }  
}
```



Visitor: example

```
public class XMLExportVisitor implements Visitor {  
    public String export(Shape shape) {  
        StringBuilder sb = new StringBuilder();  
        sb.append("<?xml version=\"1.0\" encoding=\"utf-8\"?>" + "\n");  
        sb.append(shape.accept(this));  
        return sb.toString();  
    }  
  
    public String visitCircle(Circle c) {  
        return "<circle>" + "\n" + "\t<diameter>" + c.getDiameter() + "</diameter>" + "\n" + "</circle>";  
    }  
  
    public String visitSquare(Square s) {  
        return "<square>" + "\n" + "\t<size>" + s.getSize() + "</size>" + "\n" + "</square>";  
    }  
  
    public String visitCompoundShape(CompoundShape cs) {  
        StringBuilder sb = new StringBuilder();  
        for (Shape shape : cs) {  
            String obj = shape.accept(this);  
            obj = "\t" + obj.replace("\n", "\n\t") + "\n";  
            sb.append(obj);  
        }  
        return "<compound_shape>" + "\n" + sb.toString() + "</compound_shape>";  
    }  
}
```

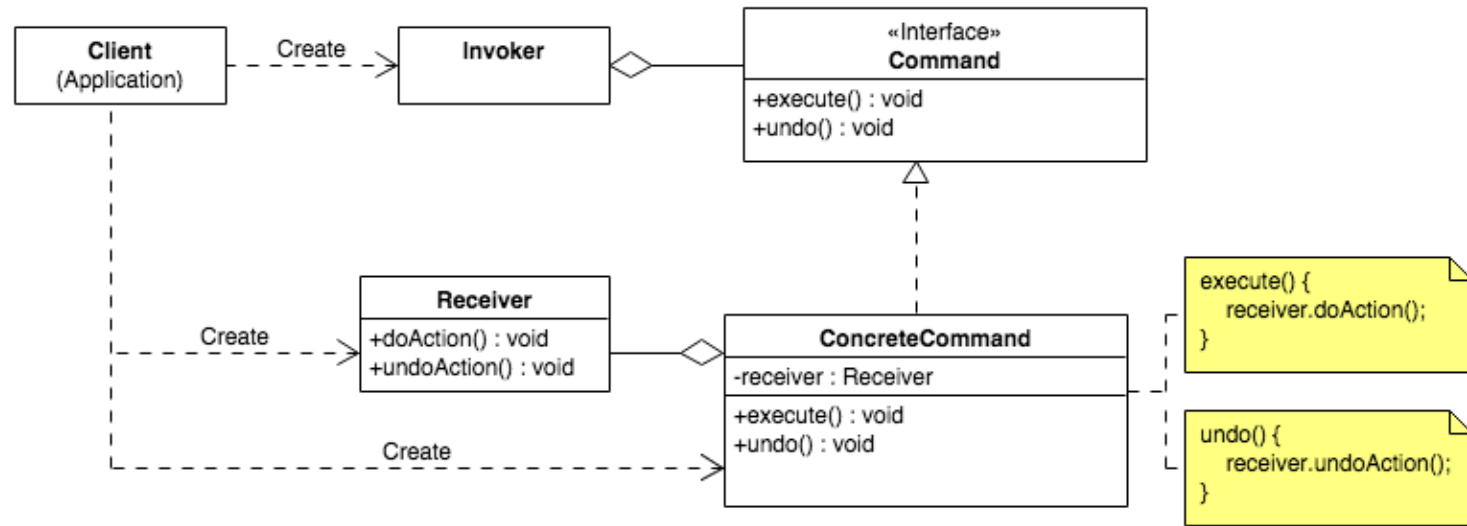


Visitor example

```
public class Main {  
    public static void main(String[] args) {  
        Shape circle= new Circle(3.0);  
        Shape square= new Square(4.0);  
        CompoundShape group= new CompoundShape();  
        group.addShape(circle);  
        group.addShape(square);  
        Shape circle2= new Circle(4.0);  
        CompoundShape group2= new CompoundShape();  
        group2.addShape(circle2);  
        group2.addShape(group);  
        XMLExportVisitor exportVisitor = new XMLExportVisitor();  
        System.out.println(exportVisitor.export(group2));  
    }  
}
```

```
<?xml version="1.0" encoding="utf-8"?>  
<compound_shape>  
    <circle>  
        <diameter>4.0</diameter>  
    </circle>  
    <compound_shape>  
        <circle>  
            <diameter>3.0</diameter>  
        </circle>  
        <square>  
            <size>4.0</size>  
        </square>  
    </compound_shape>  
</compound_shape>
```

Command Pattern



- Une commande encapsule une action à exécuter plus tard

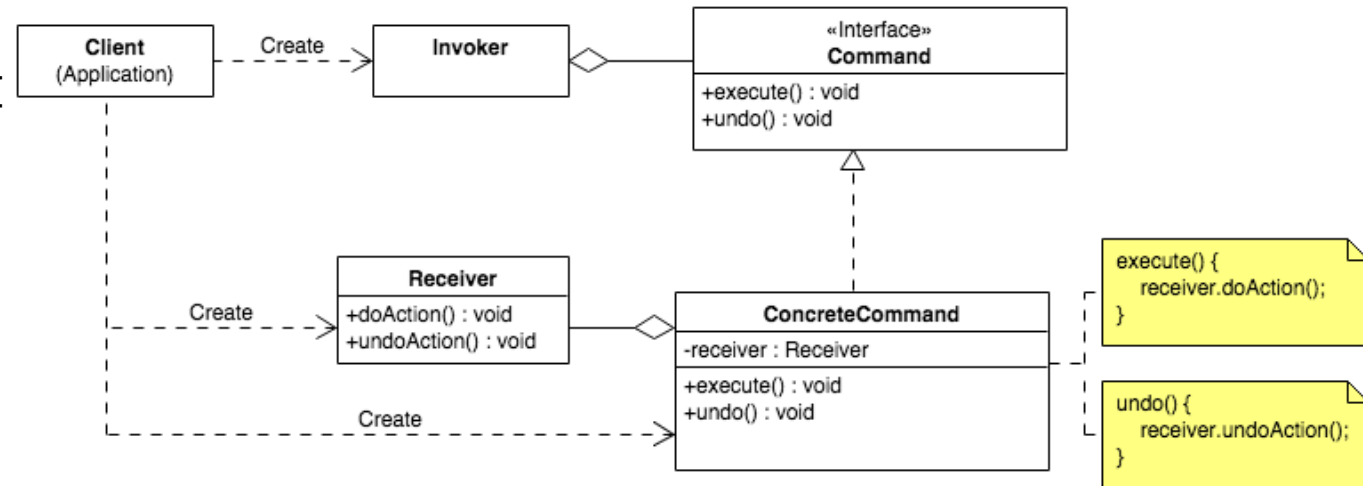
```
public class TextFile {
    private String name;
    public TextFile(String name) {this.name = name;}
    public void open() {System.out.println("Opening file " + name);}
    public void save() {System.out.println("Saving file " + name);}
    // additional text file methods (editing, writing, copying, pasting)
}
```

```
public interface TextFileOperation {
    void execute();
}
```

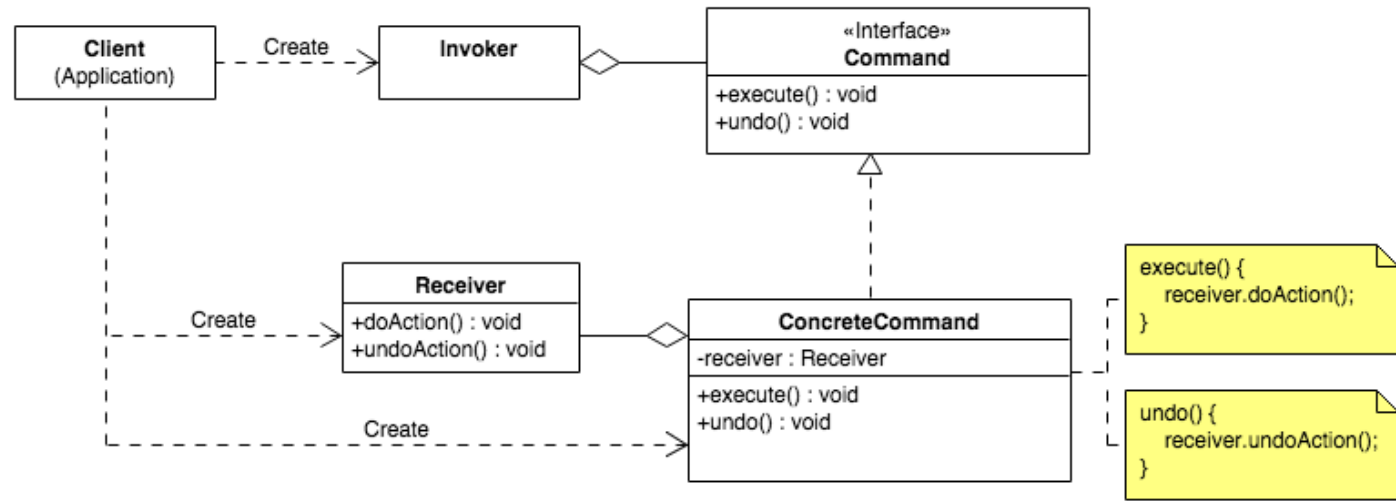
Command Pattern

```
public class OpenTextFileOperation implements TextFileOperation {  
    private TextFile textFile;  
    public OpenTextFileOperation(TextFile textFile) {  
        this.textFile = textFile;  
    }  
    @Override  
    public void execute() {textFile.open();}  
}
```

```
public class TextFileOperationExecutor {  
    private final List<TextFileOperation> textFileOperations = new ArrayList<>();  
    public void addOperation(TextFileOperation textFileOperation) {  
        textFileOperations.add(textFileOperation);  
    }  
    public void executeOperations() {  
        for (TextFileOperation tfo: textFileOperations) {tfo.execute();}  
    }  
}
```

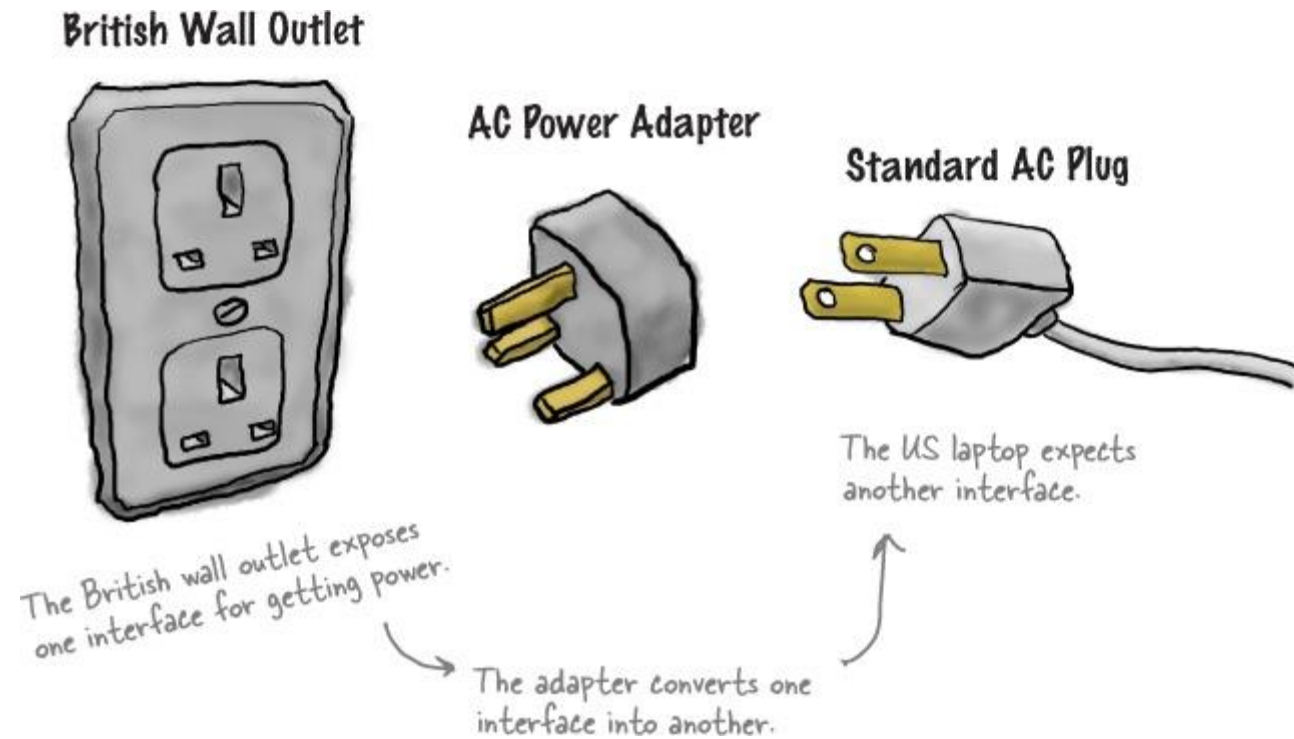


Command Pattern



```
public class Main {
    public static void main(String[] args) {
        TextFileOperationExecutor textFileOperationExecutor =
            new TextFileOperationExecutor();
        textFileOperationExecutor.addOperation(new
            OpenTextFileOperation(new TextFile("file1.txt")));
        textFileOperationExecutor.addOperation(new
            SaveTextFileOperation(new TextFile("file2.txt")));
        textFileOperationExecutor.executeOperations();
    }
}
```

Adapter



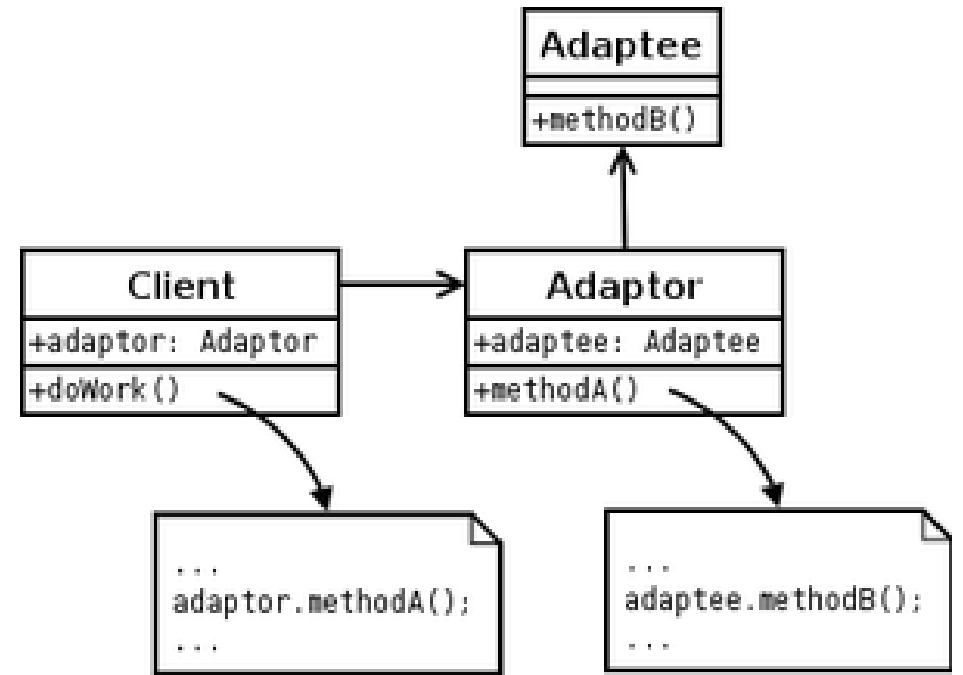
- Le pattern permet à des classes de travailler ensemble malgré leur interface incompatible

Adapter: exemple

```
public class UKElectricalSocket {  
    public void plugIn(UKPlugConnector plug) {  
        plug.provideElectricity();  
    }  
}
```

```
public interface BelgianPlugConnector {  
    public void giveElectricity() ;  
}
```

```
public class BelgianToUKPlugConnectorAdapter  
    implements UKPlugConnector {  
    private BelgianPlugConnector plug;  
    public BelgianToUKPlugConnectorAdapter(BelgianPlugConnector plug)  
    {this.plug = plug;}  
    @Override  
    public void provideElectricity() {  
        plug.giveElectricity();  
    }  
}
```



```
public class Main {  
    public static void main(String[] args) {  
        BelgianPlugConnector plugConnector =  
            new BelgianPhilipsPlugConnector();  
        UKElectricalSocket electricalSocket =  
            new UKElectricalSocket();  
        UKPlugConnector ukAdapter =  
            new BelgianToUKPlugConnectorAdapter(plugConnector);  
        electricalSocket.plugIn(ukAdapter);  
    }  
}
```

Suite du cours ...

- Le but de ce cours est de maîtriser tous les patterns
- A chaque séance sera proposé un (ou plusieurs) exercice(s) d'un des types suivant :
 - Identifier un pattern déjà implémenté dans du code.
 - Identifier un pattern pour la résolution d'un problème donné et l'implémenter
 - Identifier un pattern afin d'améliorer un code déjà existant et adapter ce code (refactoring)
- L'évaluation finale consistera en des exercices similaires:
 - Moodle accessible durant l'examen
 - Vous avez droit à une feuille recto/verso supplémentaire