DESIGN PATTERNS (Patron conception)

Creational Patterns: Used to construct objects such that they can be decoupled from their implementing system.

- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton

Structural Patterns: Used to form large object structures between many disparate objects.

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

Behavioral Patterns: Used to manage algorithms, relationships, and responsibilities between objects.

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

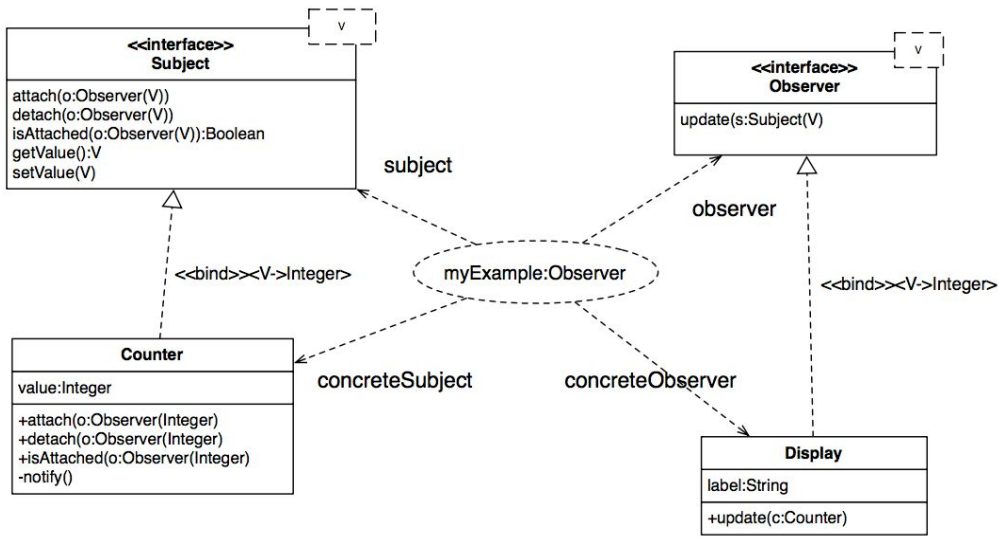
Patron: Observer

Le pattern Observateur (en anglais Observer) définit une relation entre objets de type un-à-plusieurs, de façon que, si un objet change d'état, tous ceux qui en dépendent en soient informés et mis à jour automatiquement.

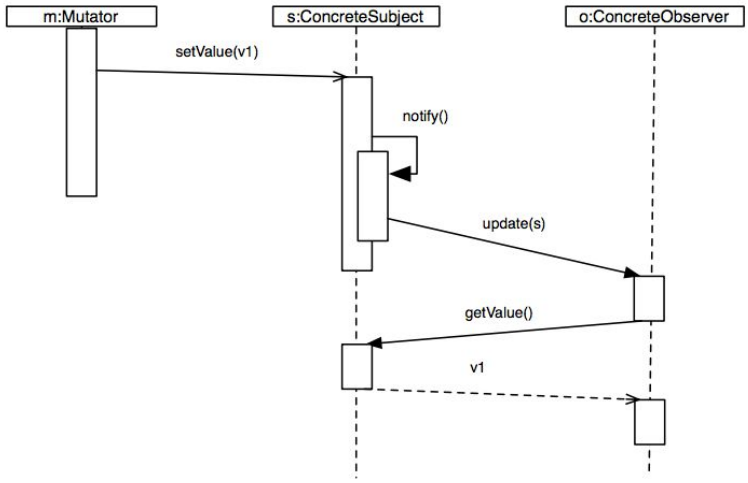
→ **Motivation:** Propagation des changements d'un objet aux autres
→ **Intention:** Il existe des objets qui doivent avoir leur état synchronisé avec l'état des autres objets.
→ **Participants / Responsabilités:**

- Subject (interface): manages observers' subscription (interface plus storage of subscriptions)
- Observer (interface): provides an interface to receive update notification from observers
- Mutator (interface): the outside, the reason why subject states change
- Concrete subject: stores a data state, provides R/U operations for it. Must notify observers when its state changes.
- Concrete observer: Has a state that depends on the subject's state. Provides a method for the operation that the subject can call to notify changes

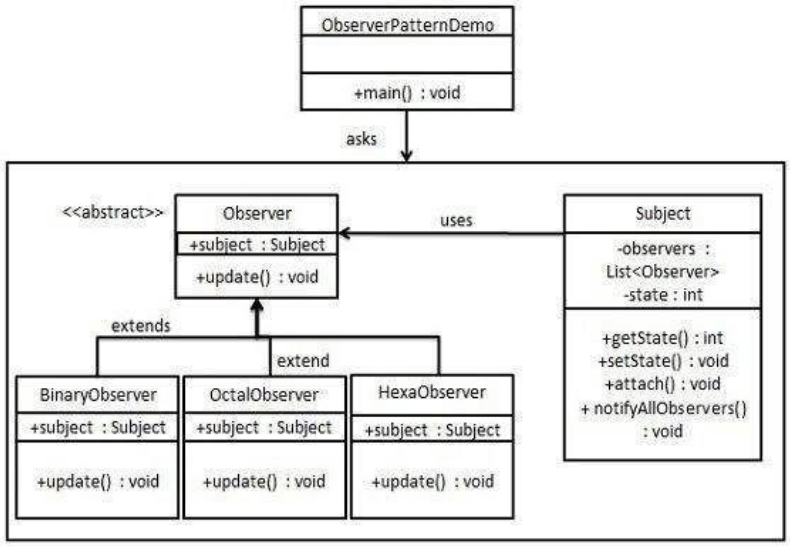
→ **Diagramme Classes:**



→ **Diagramme Sequence (collaboration):**



→ **Example:**



Step 1 Create Subject class.

Subject.java

```
import java.util.ArrayList;
import java.util.List;


public class Subject {

   private List<Observer> observers = new ArrayList<Observer>();
   private int state;

   public int getState() {
      return state;
   }

   public void setState(int state) {
      this.state = state;
      notifyAllObservers();
   }

   public void attach(Observer observer){
      observers.add(observer);
   }
```

```
   public void notifyAllObservers(){
      for (Observer observer : observers) {
         observer.update();
      }
   }
}
```

Step 2 Create Observer class.

Observer.java

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

Step 3 Create concrete observer classes

BinaryObserver.java

```
public class BinaryObserver extends Observer{

   public BinaryObserver(Subject subject){
      this.subject = subject;
      this.subject.attach(this);
   }

   @Override
   public void update() {
      System.out.println( "Binary String: " + Integer.toBinaryString( subject.getState() ) );
   }
}
```

OctalObserver.java

```
public class OctalObserver extends Observer{

   public OctalObserver(Subject subject){
      this.subject = subject;
      this.subject.attach(this);
   }

   @Override
   public void update() {
     System.out.println( "Octal String: " + Integer.toOctalString( subject.getState() ) );
   }
}
```

Step 4 Use Subject and concrete observer objects.

ObserverPatternDemo.java

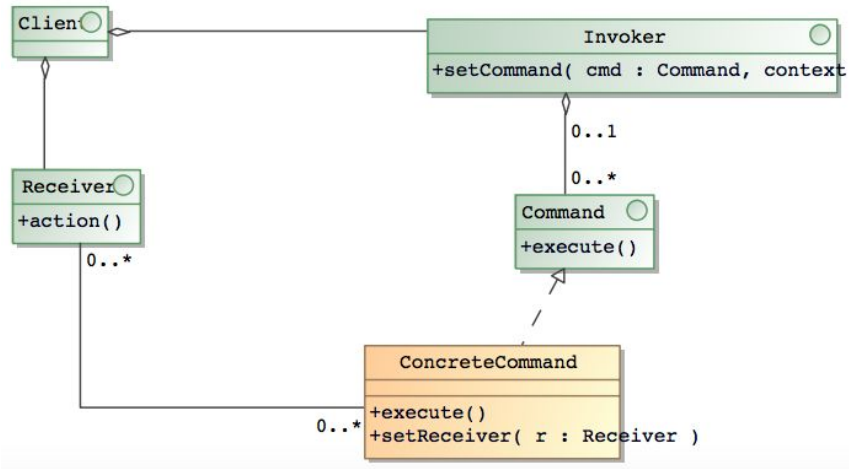
```
public class ObserverPatternDemo {
   public static void main(String[] args) {
      Subject subject = new Subject();

      new HexaObserver(subject);
      new OctalObserver(subject);
      new BinaryObserver(subject);

      System.out.println("First state change: 15");
      subject.setState(15);
      System.out.println("Second state change: 10");
      subject.setState(10);
   }
}
```

Patron: Command

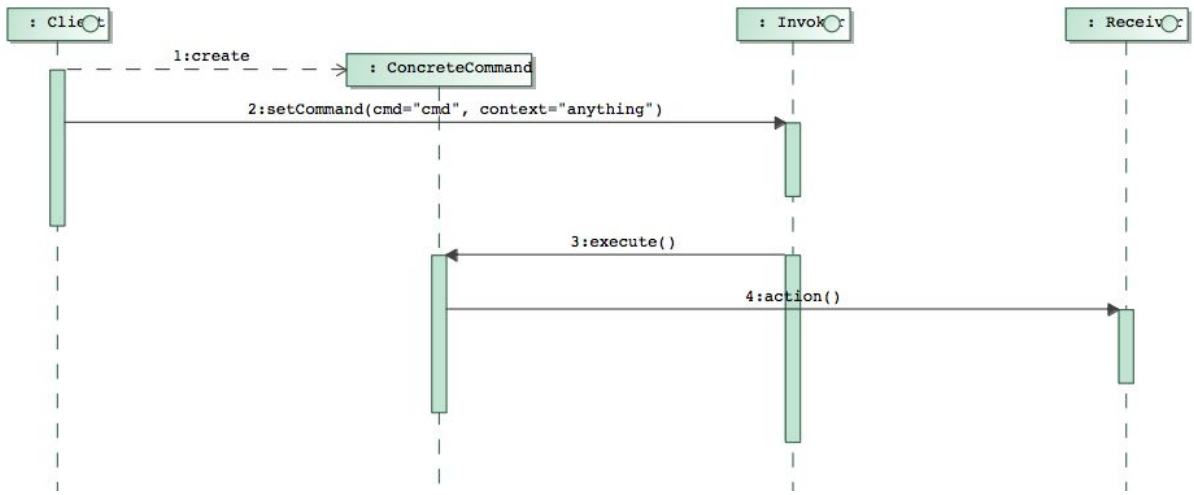
→ **Motivation:** Reify an operation concept into an object
→ **Intention:** Often one needs to choose an operation and call it later
→ **Participants / Responsabilités:**

- command (interface): Define operations common to all commands, for invocation. act as a relay(paso) between invoker and receiver. minimum: execute()
- invoker (interface): when appropriate requests execution from a command.
- receiver (interface): performs the task upon request by a command execution.
- concrete command: knows which receiver to use and what operation to call. Implements the command operation to forward calls to receivers.
- client: creates and configure concrete commands. Register commands with the invoker

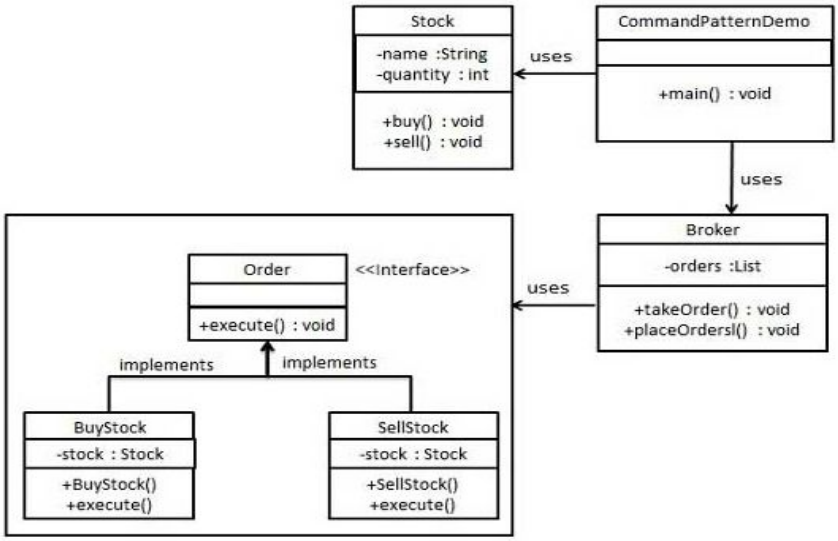
→ **Diagramme Classes:**



→ **Diagramme Sequence:**



→ **Example:**



Step 1 Create a command interface.
Order.java

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

Step 2 Create a request class.
Stock.java

```
public class Stock {

   private String name = "ABC";
   private int quantity = 10;

   public void buy(){
      System.out.println("Stock [ Name: "+name+",
         Quantity: " + quantity +" ] bought");
   }
   public void sell(){
      System.out.println("Stock [ Name: "+name+",
         Quantity: " + quantity +" ] sold");
   }
}
```

Step 3 Create concrete classes implementing the *Order* interface.
BuyStock.java

```
public class BuyStock implements Order {
   private Stock abcStock;

   public BuyStock(Stock abcStock){
      this.abcStock = abcStock;
   }

   public void execute() {
      abcStock.buy();
   }
}
```

SellStock.java

```
public class SellStock implements Order {
   private Stock abcStock;
```

```
   public SellStock(Stock abcStock){
      this.abcStock = abcStock;
   }

   public void execute() {
      abcStock.sell();
   }
}
```

Step 4 Create command invoker class.
Broker.java

```
import java.util.ArrayList;
import java.util.List;
public class Broker {
   private List<Order> orderList = new ArrayList<Order>();

   public void takeOrder(Order order){
      orderList.add(order);
   }

   public void placeOrders(){
      for (Order order : orderList) {
         order.execute();
      }
      orderList.clear();
   }
}
```

Step 5 Use the Broker class to take and execute commands.
CommandPatternDemo.java

```
public class CommandPatternDemo {
   public static void main(String[] args) {
      Stock abcStock = new Stock();

      BuyStock buyStockOrder = new BuyStock(abcStock);
      SellStock sellStockOrder = new SellStock(abcStock);

      Broker broker = new Broker();
      broker.takeOrder(buyStockOrder);
      broker.takeOrder(sellStockOrder);

      broker.placeOrders();
   }
}
```

PATTERNS DE DÉLÉGATION

Usages de la délégation

- Extension d'opération
- Extension de méthode
- Joue un rôle dans la réification

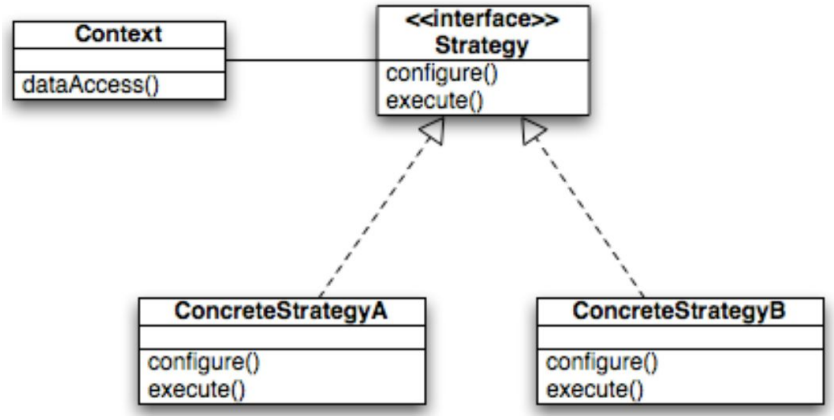
Patron: Strategy

→ **But:** Permettre de mettre en oeuvre des algorithmes différents avec un choix dynamique de la mise en oeuvre.
→ **Analogie:** Permet de remplacer les "pointeurs de fonction"

→ **Participants / Responsabilités:**

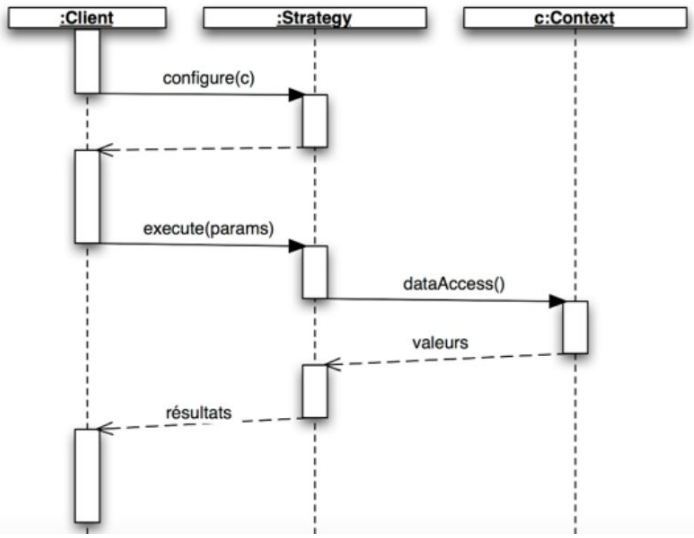
- Strategy(Inteface): Définit une interface pour configurer l'algorithme (paramètres) et l'exécuter.
- Context: Désigne l'algorithme concret en vigueur. Peut contenir des données pour l'algorithme.

→ **Diagramme Classes:**

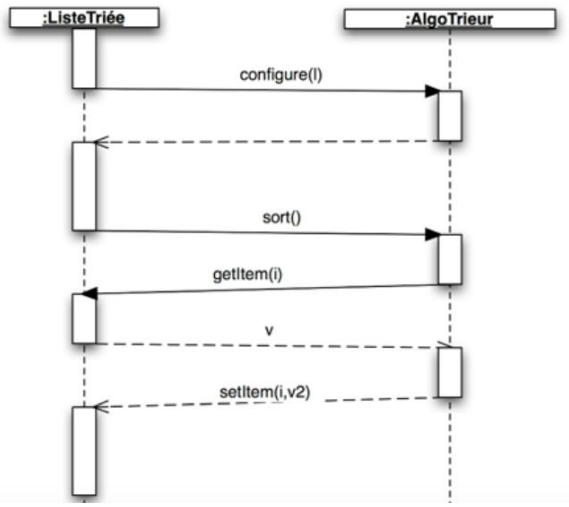


Il est possible de séparer: la demande d'exécution et la gestion des données. Dans ce cas on fait apparaître un rôle Client par exemple:

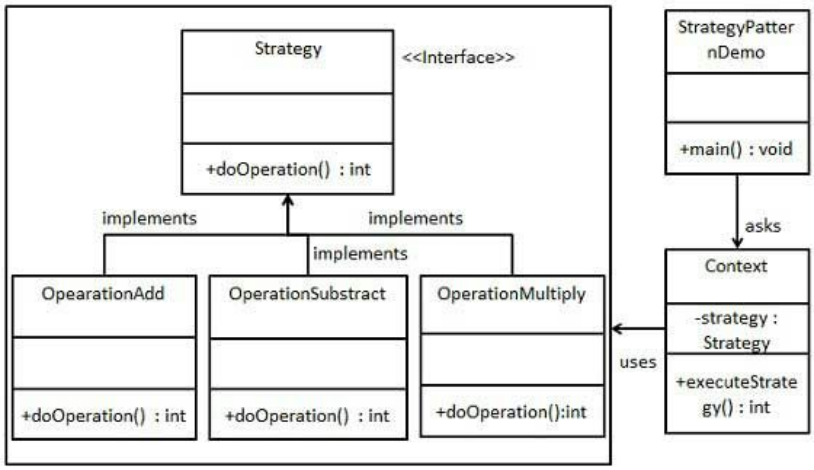
→ **Diagramme Sequence (avec client):**



→ **Diagramme Sequence (SANS client):**



→ **Exemple:**



Step 1 Create an interface.

Strategy.java

```
public interface Strategy {
   public int doOperation(int num1, int num2);
}
```

Step 2 Create concrete classes implementing the same interface.

OperationAdd.java

```
public class OperationAdd implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 + num2;
   }
}
```

OperationMultiply.java

```
public class OperationMultiply implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 * num2;
   }
}
```

Step 3 Create Context Class.

Context.java

```
public class Context {
   private Strategy strategy;

   public Context(Strategy strategy){
      this.strategy = strategy;
   }

   public int executeStrategy(int num1, int num2){
      return strategy.doOperation(num1, num2);
   }
}
```

Step 4 Use the Context to see change in behaviour when it changes its Strategy.

StrategyPatternDemo.java

```
public class StrategyPatternDemo {
   public static void main(String[] args) {
      Context context = new Context(new OperationAdd());
      System.out.println("10 + 5 = " + context.executeStrategy(10, 5));
```

```
        context = new Context(new OperationSubstract());
        System.out.println("10 - 5 = " + context.executeStrategy(10, 5));

        context = new Context(new OperationMultiply());
        System.out.println("10 * 5 = " + context.executeStrategy(10, 5));
    }
}
```

Patron: TEMPLATE METHOD

→ **Motivation:**
Fournir un point d'extension pour une mise en œuvre dans un algorithme
– par exemple une fonction de comparaison
– le mécanisme de base est ici l'héritage

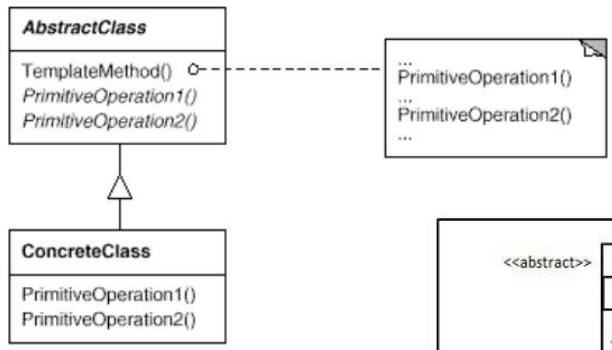
!!! Ce n'est donc pas de la délégation

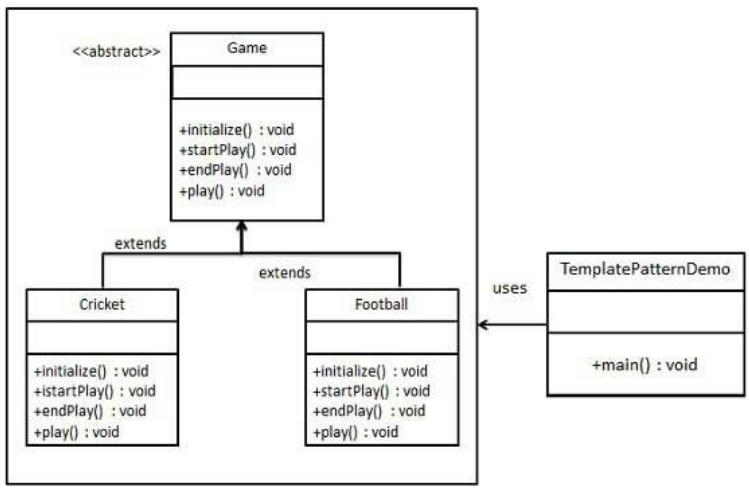
→ **Participants / Responsabilités:**

- AbstractClass: Définit un algorithme «à trous». Chaque trou est l'appel à une opération interne abstraite .
- ConcreteClass: étend AbstractClass en fournissant des méthodes pour les opérations abstraites.
- Composable avec Strategy:

Comme évoqué dans la partie Strong Objects, l'héritage doit être soigneusement documenté.
Dans Template Method le protocole que chaque sous-classe doit mettre en œuvre est nécessairement explicité
On pourrait même dire que tout héritage de méthode doit suivre le PC Template Method

→ **Diagramme Classes:**





→ **Example:**
Step 1 Create an abstract class with a template method being final.

Game.java

```
public abstract class Game {
   abstract void initialize();
   abstract void startPlay();
   abstract void endPlay();

   //template method
   public final void play(){

      //initialize the game
      initialize();
      //start game
      startPlay();
      //end game
      endPlay();
   }
}
```

Step 2 Create concrete classes extending the above class.

Cricket.java

```
public class Cricket extends Game {
@Override
   void endPlay() {
      System.out.println("Cricket Game Finished!");
   }

   @Override
   void initialize() {
      System.out.println("Cricket Game Initialized! Start playing.");
   }
   @Override
   void startPlay() {
      System.out.println("Cricket Game Started. Enjoy the game!");}}
```

Football.java

```
public class Football extends Game {
   @Override
   void endPlay() {
      System.out.println("Football Game Finished!");
   }

   @Override
   void initialize() {
      System.out.println("Football Game Initialized! Start playing.");
   }

   @Override
   void startPlay() {
      System.out.println("Football Game Started. Enjoy the game!");
   }
}
```

Step 3 Use the *Game*'s template method play() to demonstrate a defined way of playing game.

TemplatePatternDemo.java

```
public class TemplatePatternDemo {
   public static void main(String[] args) {
      Game game = new Cricket();
      game.play();
      System.out.println();
      game = new Football();
      game.play();
   }
}
```

Patron: PROXY

→ **Motivation:** Fournir un objet qui agit comme une «doublure». Cette doublure est plus économique / plus simple,...Cette doublure a l'interface de l'original. Elle ne possède que certaines propriétés. Elle rend les autres services par délégation.

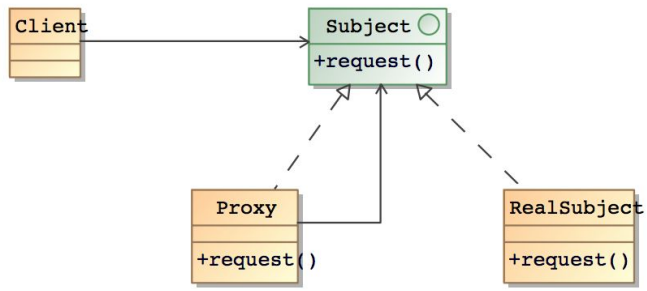
→ **Intention:** Provide a surrogate or placeholder for another object to control access to it. Use an extra level of indirection to support distributed, controlled, or intelligent access. Add a wrapper and delegation to protect the real component from undue complexity.

→ **Participants / Responsabilités:**

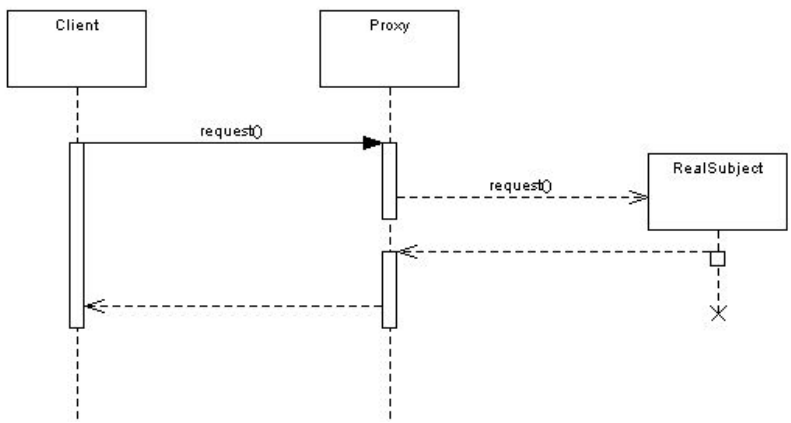
- Subject: l'interface de définition du service, des propriétés.
- RealSubject: Le «gros objet» qui détient les vrais attributs et les vraies méthodes
- Proxy: fait semblant d'être un gros objet

→ **Avantages**: Le principe général permet d'ajouter des services «méta » dans le Proxy.
→ **Inconvénients**: Une indirection peut rendre l'identité de référence incorrecte

→ **Diagramme Classes:**



→ **Diagramme Sequence:**



Patron: Adapter

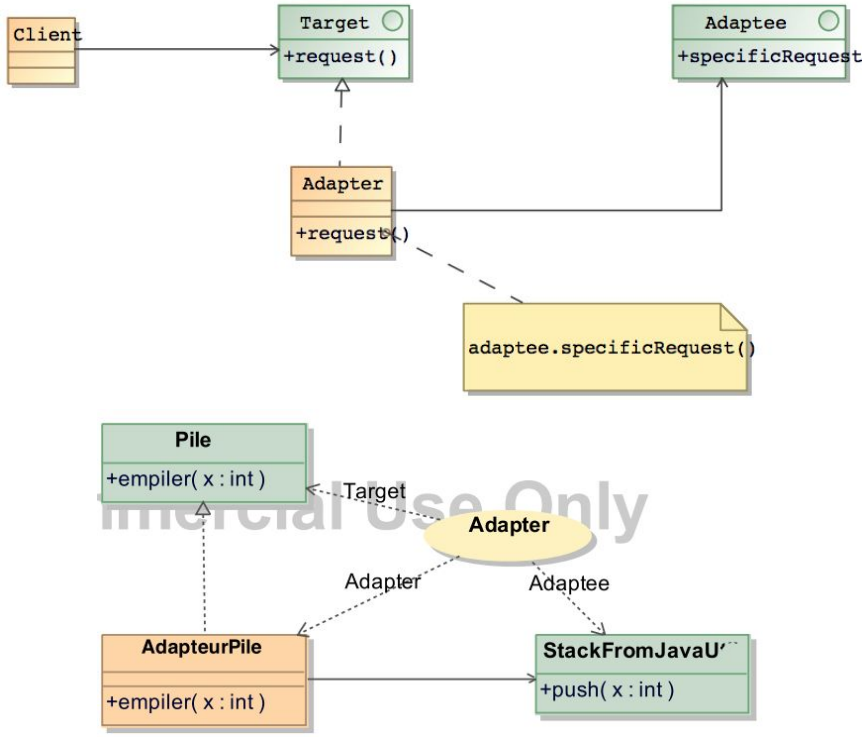
→ **Motivation:**

Changer des opérations en réemployant des méthodes existantes (Catégorie: structure). Les opérations font en effet partie de la structure.

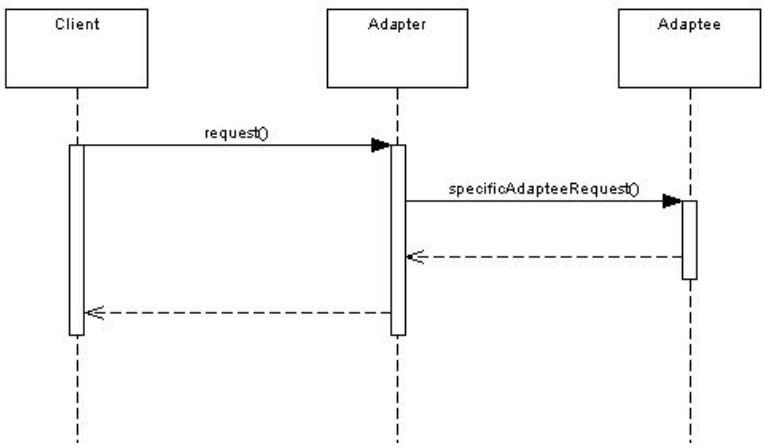
→ **Participants / Responsabilités:**

- Target: l'interface qui définit les nouvelles opérations.
- Adaptee: les anciennes opérations et leurs méthodes accédé par délégation (variante: accès par héritage)
- Client: Accède aux méthodes en employant les nouvelles opérations. Ignore l'existence des anciennes opérations.
- Adapter: met en œuvre l'interface Target et réalise les méthodes par délégation. Effectue toutes les conversions nécessaires.

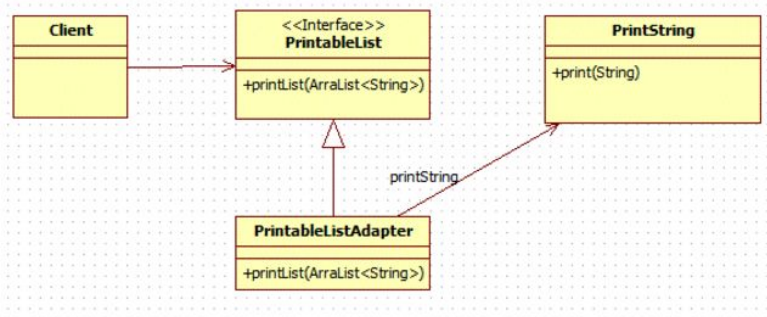
→ **Diagramme Classes:**



→ **Diagramme Sequence:** Clients call operations on the adapter instance. In turn adapter calls adaptee operations that carry out the request.



→ **Example:**



Java Code: Consider that we have a third party library that provides print string functionality through PrintString class. This is our **Adaptee**. I know this is silly assumption but lets go with it for now.

PrintString.java(Adaptee) :

```
package org.arpit.javapostsforlearning.designpatterns;

public class PrintString {
        public void print(String s){
                System.out.println(s);
        }
}
```

Client deals with `ArrayList<String>` but not with string.We have provided a `PrintableList` interface that expects the client input. This is our **target**.

PrintableList.java(Target)

```
package org.arpit.javapostsforlearning.designpatterns;
import java.util.ArrayList;

public interface PrintableList {
        void printList(ArrayList<String> list);
}
```

Let's assume we can not change it now. Finally we have PrintableListAdapter class which will implement PrintableList interface and will deal with our adaptee class.

PrintableListAdapter.java(Adapter)

```
package org.arpit.javapostsforlearning.designpatterns;
import java.util.ArrayList;

public class PrintableListAdapter implements PrintableList{
        public void printList(ArrayList<String> list) {
                //Converting ArrayList<String> to String so that we can pass String to adaptee
class
                String listString = "";
                for (String s : list){
                    listString += s + "\t";
                }
                // instantiating adaptee class
                PrintString printString=new PrintString();
                ps.print(listString);  }  }
```

AdapterDesignPatternMain.java

```
package org.arpit.javapostsforlearning.designpatterns;

import java.util.ArrayList;
public class AdapterDesignPatternMain {
        public static void main(String[] args){
                ArrayList<String> list=new  ArrayList<String>();
```

```
                list.add("one");
                list.add("two");
                list.add("three");
                PrintableList pl=new PrintableListAdapter();
                pl.printList(list);
        }
}
```

Patron: DECORATOR

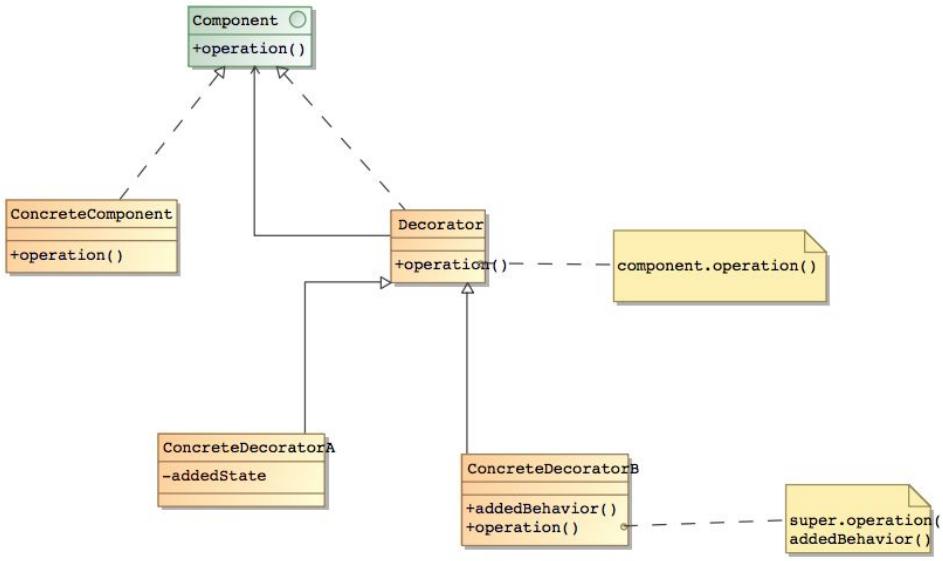
→ **Motivation:** Étendre dynamiquement les possibilités d'une objet

- au moyen de nouvelles méthodes
- alternative intéressante à l'extension statique de type (héritage)
- structure

→ **Participants / Responsabilités:**

- **Component**: interface définissant les opérations
- **ConcreteComponent**: la mise en œuvre initiale des opérations
- **Decorator**: une classe abstraite qui gère la délégation par défaut
- **ConcreteDecorator**: une mise en œuvre étendue par l'ajout d'atributs, de méthodes. Emploie l'appel à super pour activer la délégation. Emploie l'appel à des opérations locales pour ajouter du comportement.

→ **Diagramme Classes:**



→ **Exemple (Prof)**

```
public interface CoffeeMachine {
        /**
        * Provides a recycled cup from garbage collector
        * @return   a shiny cup
        */
        Cup  provideCup();

        /**
        * Fills a cup to max capacity
        * @param cup
        */
        void pourCoffee(Cup cup);
```

```
}
public class BasicCoffeeMachine implements CoffeeMachine {
        /**
        * Provides a recycled cup from garbage collector
        * @return a shiny cup
        */
        @Override
        public Cup provideCup() {

        return new BasicCup(10.0);
        }

        /**
        * Fills a cup to max capacity
        *
        * @param cup
        */
        @Override
        public void pourCoffee(Cup cup) {
        Logger.getGlobal().severe("Internal machine jam");
        }
}

public class CoffeeMachineDecorator implements CoffeeMachine {

        private final CoffeeMachine coffeeMachineDelegate;

        public CoffeeMachineDecorator(CoffeeMachine coffeeMachineDelegate) {
        this.coffeeMachineDelegate = coffeeMachineDelegate;
        }

        /**
        * Provides a recycled cup from garbage collector
        * @return a shiny cup
        */
        @Override
        public Cup provideCup() {
        return coffeeMachineDelegate.provideCup();
        }

        /**
        * Fills a cup to max capacity
        * @param cup
        */
        @Override
        public void pourCoffee(Cup cup) {
        coffeeMachineDelegate.pourCoffee(cup);
        }
}


public interface AdvancedCoffeeMachine extends CoffeeMachine {

        /**
        * Puts one piece of sugar in cup
        * @param cup   the sugar receptacle
        */
        public void putOneSugarPieceInCup(Cup cup);

}

public     class     AdvancedCoffeeMachineImpl     extends     CoffeeMachineDecorator     implements
AdvancedCoffeeMachine {

        public AdvancedCoffeeMachineImpl(CoffeeMachine coffeeMachine) {
                super(coffeeMachine);
        }
        /**
        * Puts one piece of sugar in cup
        * @param cup the sugar receptacle
        */
```

```
        @Override
        public void putOneSugarPieceInCup(Cup cup) {
        // TODO
        }
}

public interface Cup {

        double getCapacityInCm3();
        double getSugarAmountInGrams();
        /**
        * Adds more sugar in the cup.
        *
        * @param moreSugar amount to add, must be > 0.0
        * @throws java.lang.IllegalArgumentException if amount < 0.0
        */
        void addSugarAmountInGrams(double moreSugar);
}

public class BasicCup implements Cup {

        private final double capacityInCm3;
        private double currentSugarAmountInGrams;

        public BasicCup(double capacityInCm3) {
        this.capacityInCm3 = capacityInCm3;
        }

        @Override
        public double getCapacityInCm3() {
        return capacityInCm3;
        }

        @Override
        public double getSugarAmountInGrams() {
        return currentSugarAmountInGrams;
        }

        /**
        * Adds more sugar in the cup.
        *
        * @param moreSugarInGrams amount to add, must be > 0.0
        * @throws java.lang.IllegalArgumentException if amount < 0.0
        */
        @Override
        public void addSugarAmountInGrams(double moreSugarInGrams) {

        if(moreSugarInGrams < 0.0) {
                throw new IllegalArgumentException("negative sugar amount");
        }
        currentSugarAmountInGrams += moreSugarInGrams;
        }
}


public class CoffeeMachineTest {

        private CoffeeMachine sampleCoffeeMachine;
        private Cup sampleCup;

        @Before
        public void setUp() throws Exception {

        sampleCoffeeMachine = new BasicCoffeeMachine();

        }

        @Test
        public void testProvideCup() throws Exception {
        sampleCup = sampleCoffeeMachine.provideCup();
        Assert.assertEquals(10.0, sampleCup.getCapacityInCm3(),10.0e-5);
```

```
        }

        @Test
        public void testPourCoffee() throws Exception {
        Assert.fail("testPourCoffee not implemented");
        }
}
```

Patron: Façade

→ **Intention:** Rassembler des interfaces en une seule. Masquer des éléments d'un système.
Facade defines a higher-level interface that makes the subsystem easier to use.

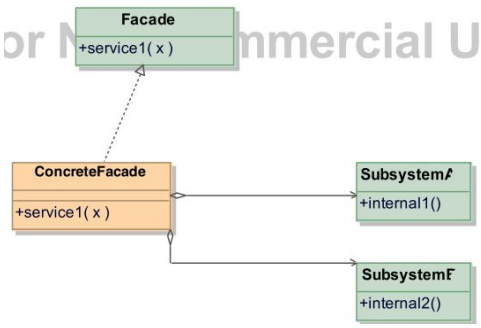
Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.

Like the adapter pattern, the Facade **can be used to hide the inner workings of a third party library,** or some legacy code. All that the client needs to do is interact with the Facade, and not the subsystem that it is encompassing.

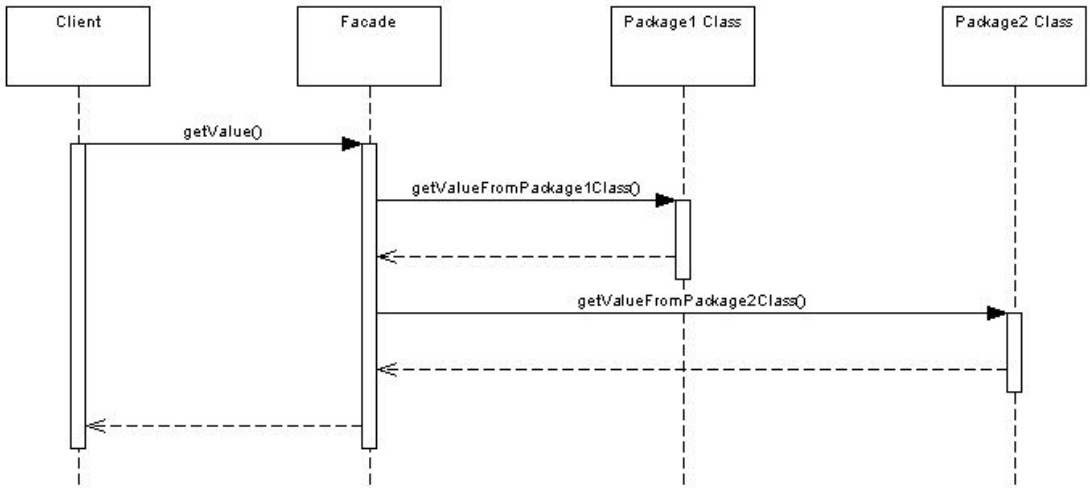
→ **Participants / Responsabilités:**

- Façade: l'interface qui définit les opérations visibles de l'extérieur.
- ConcreteFacade: la mise en œuvre de la délégation.
- Subsystem: les éléments (interfaces, classes) qui composent le sous-système

→ **Diagramme Classes:**



→ **Diagramme Sequence:**



Patron: Bridge

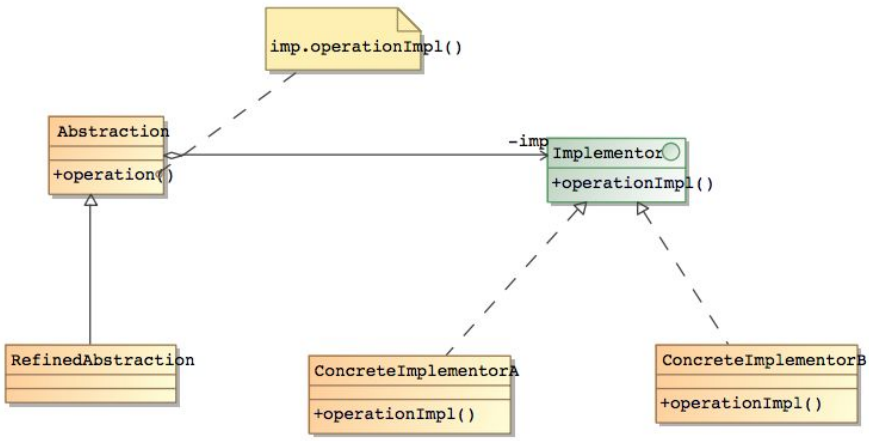
→ **Motivation:** Séparer définition des opérations (interface) et définitions des méthodes en les connectant par délégation.
Permettre une spécialisation séparée des interfaces et des mises en œuvre

→ **Intention:** Il existe des objets qui doivent avoir leur état synchronisé avec l'état des autres objets.

→ **Participants / Responsabilités:**

- **Abstraction**: Définition des opérations et mise en œuvre du mécanisme de délégation
- **Refined abstraction**: Extension des opérations.
- **Implementor**: Interface pour la mise en œuvre, peut avoir des opérations différentes.
- Concrete Implementor: Les méthodes mettant en œuvre Implementor

→ **Diagramme Classes:**



Synthèse des usages

- ➔ Emploi de la délégation pour cacher une structure: Proxy, Adapter
- ➔ Emploi de la délégation pour changer une méthode: Strategy, Decorator, Template method, Bridge