Design Patterns
Part 8

Mohamed Youssfi

Laboratoire Signaux Systèmes Distribués et Intelligence Artificielle (SSDIA)

ENSET, Université Hassan II Casablanca, Maroc

Email : med@youssfi.net

Supports de cours : http://fr.slideshare.net/mohamedyoussfi9

Chaîne vidéo : http://youtube.com/mohamedYoussfi

Recherche : http://www.researchgate.net/profile/Youssfi_Mohamed/publications

Patterns :

- Stat
- Template Method
- Command
- Mediator

Pattern Stat

- Le comportement des méthodes d'un objet dépendent souvent de l'état de celui-ci.
- Cet état est en général représente par les attributs de l'objet.
- Cependant, plus l'objet peut prendre d'états comportementaux distincts, plus la structuration de l'objet est complexe.
- Le design pattern "état" est la meilleure solution pour bien structurer et représenter les différents états d'un objet ainsi que les transitions entre ces états sous une forme orientée objet.

Exemple de problème

```
interface Avion {
  void sortirDuGarage () ;
  void decoller () ;
  void atterir () ;
  void entrerAuGarage () ;
}
```

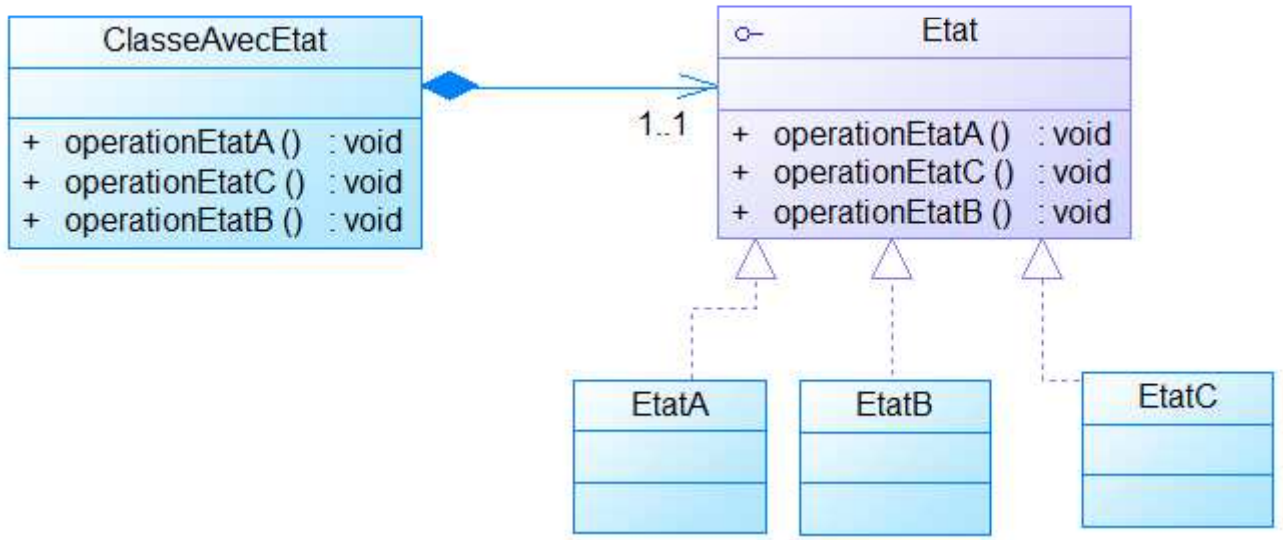
- Un avion peut être dans l'un des trois états suivants :
  - soit dans le garage,
  - soit sur la piste,
  - soit en l'air.
- Lorsqu'il est dans le garage, la méthode ***sortirDuGarage*** permet de passer dans l'état "**sur la piste**" .
- Lorsqu'il est sur la piste, la méthode ***entrerAuGarage*** permet de passer dans l'état "**dans le garage**"
- La méthode ***décoller*** permet de passer dans l'état "**en l'air**".
- Lorsqu'il est en l'air, la méthode ***atterrir*** permet de passer dans l'état "**sur la piste**".
- Les autres combinaisons Etats - Méthodes génèrent une erreur comme par exemple, invoquer décoller lorsque l'avion est dans le garage.

Pattern Stat

- Catégorie :
  - Comportement
- Objectif du pattern
  - *Changer le comportement d'un objet selon son état interne.*
- Résultat :
  - Le Design Pattern permet d'isoler les algorithmes propres à chaque état d'un objet.

Design pattern Etat ou State

- Le pattern Etat permet de déléguer le comportement d'un objet dans un autre objet. Cela permet de changer le comportement de l'objet en cours d'exécution et de simuler un changement de classe.



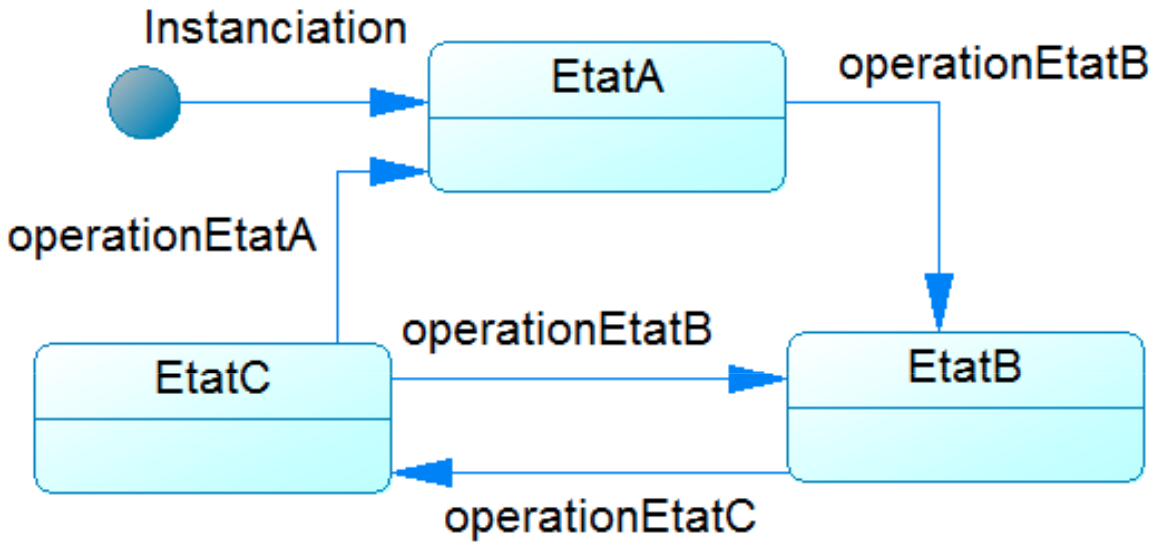
Raison d'utilisation

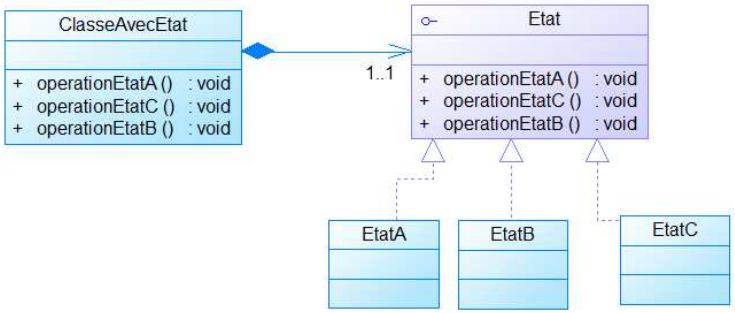
- Un objet a un fonctionnement différent selon son état interne. Son état change selon les **méthodes appelées.**
- Cela peut être un document informatique. Il a comme fonctions ouvrir, modifier, sauvegarder ou fermer. Le
- comportement de ces méthodes change selon l'état du document.
- Les différents états internes sont chacun représenté par une **classe état (ouvert, modifié, sauvegardé et fermé).**
- Les états possèdent des méthodes permettant de réaliser les opérations et de changer d'état (ouvrir, modifier,sauvegarder et fermer). Certains états bloquent certaines opérations (modifier dans l'état fermé).
- L'objet avec état maintient une référence vers l'état actuel. Il présente les opérations à la partie cliente.

Responsabilités

- **ClasseAvecEtat** : est une classe avec état. Son comportement change en fonction de son état. La partie changeante de son comportement est déléguée à un objet Etat.
- **Etat** : définit l'interface d'un comportement d'un état.
- **EtatA**, **EtatB** et **EtatC** : sont des sous-classes concrètes de l'interface Etat. Elles implémentent des méthodes qui sont associées à un Etat.

Diagramme d'état transition





Implémentation

/* Etat.java */

```
public abstract class Etat {
  protected ClasseAvecEtat classeAvecEtat;

  public Etat(ClasseAvecEtat classeAvecEtat) {
  this.classeAvecEtat = classeAvecEtat;
  }
   public abstract void operationEtatA();
   public abstract void operationEtatB();
   public abstract void operationEtatC();
   public abstract void doAction();
}
```

Implémentation

/* ClasseAvecEtat.java */

```
public class ClasseAvecEtat {
  private Etat etat;
   public ClasseAvecEtat() {  etat=new EtatA(this); }
   public void operationEtatA() {  etat.operationEtatA();}
  public void operationEtatB() { etat.operationEtatB(); }
  public void operationEtatC() { etat.operationEtatC(); }
  public void doAction() { etat.doAction(); }
  public Etat getEtat() { return etat; }
  public void setEtat(Etat etat) { this.etat = etat; }
}
```

Implémentation

/* EtatA.java */

```
public class EtatA extends Etat{
public EtatA(ClasseAvecEtat classeAvecEtat) {super(classeAvecEtat); }
@Override
public void operationEtatA() {
   System.out.println("Classe déjà dans l'état A");
}
@Override
public void operationEtatB() {
   classeAvecEtat.setEtat(new EtatB(classeAvecEtat));
   System.out.println("Changement d'état de A=>B");
}
@Override
public void operationEtatC() {
   System.out.println("Impossible de passer de A =>C");
}
@Override
public void doAction() { System.out.println("Etat courant : A"); }
 }
```

Implémentation

/* EtatA.java */

```
public class EtatB extends Etat{
public EtatB(ClasseAvecEtat classeAvecEtat) { super(classeAvecEtat);}
@Override
public void operationEtatA() {
   System.out.println("Pas de possible de passer de B vers A");
}
@Override
public void operationEtatB() {
   System.out.println("Déjà dans l'état B");
}
@Override
public void operationEtatC() {
   classeAvecEtat.setEtat(new EtatC(classeAvecEtat));
   System.out.println("Changement d'état de B vers C");
}
@Override
public void doAction() { System.out.println("Etat courant : B"); }
 }
```

Implémentation

/* EtatC.java */

```
public class EtatC extends Etat{
public EtatC(ClasseAvecEtat classeAvecEtat) { super(classeAvecEtat); }
@Override
public void operationEtatA() {
   System.out.println("Changement d'état de C vers A");
   classeAvecEtat.setEtat(new EtatA(classeAvecEtat));
}
@Override
public void operationEtatB() {
   System.out.println("Changement d'état de C vers B");
   classeAvecEtat.setEtat(new EtatB(classeAvecEtat));
}
@Override
public void operationEtatC() {
   System.out.println("Déjà dans l'état C");
}
@Override
public void doAction() { System.out.println("Etat courant : C"); }
 }
```

Implémentation

/* Application.java */

```
public class Application {
public static void main(String[] args) {
   ClasseAvecEtat obj=new ClasseAvecEtat(); obj.doAction();
   System.out.println("-------------");
   obj.operationEtatA(); obj.doAction();System.out.println("-------------");
   obj.operationEtatC(); obj.doAction();System.out.println("-------------");
   obj.operationEtatB(); obj.doAction();System.out.println("-------------");
   obj.operationEtatC(); obj.doAction();System.out.println("-------------");
   obj.operationEtatA(); obj.doAction();System.out.println("-------------");
}
}
```

```
Etat courant : A
-------------
Classe déjà dans l'état A
Etat courant : A
-------------
Impossible de passer de A =>C
Etat courant : A
-------------
```

```
Changement d'état de A=>B
Etat courant : B
-------------
Changement d'état de B vers C
Etat courant : C
-------------
Changement d'état de C vers A
Etat courant : A
-------------
```

Pattern
Template Method

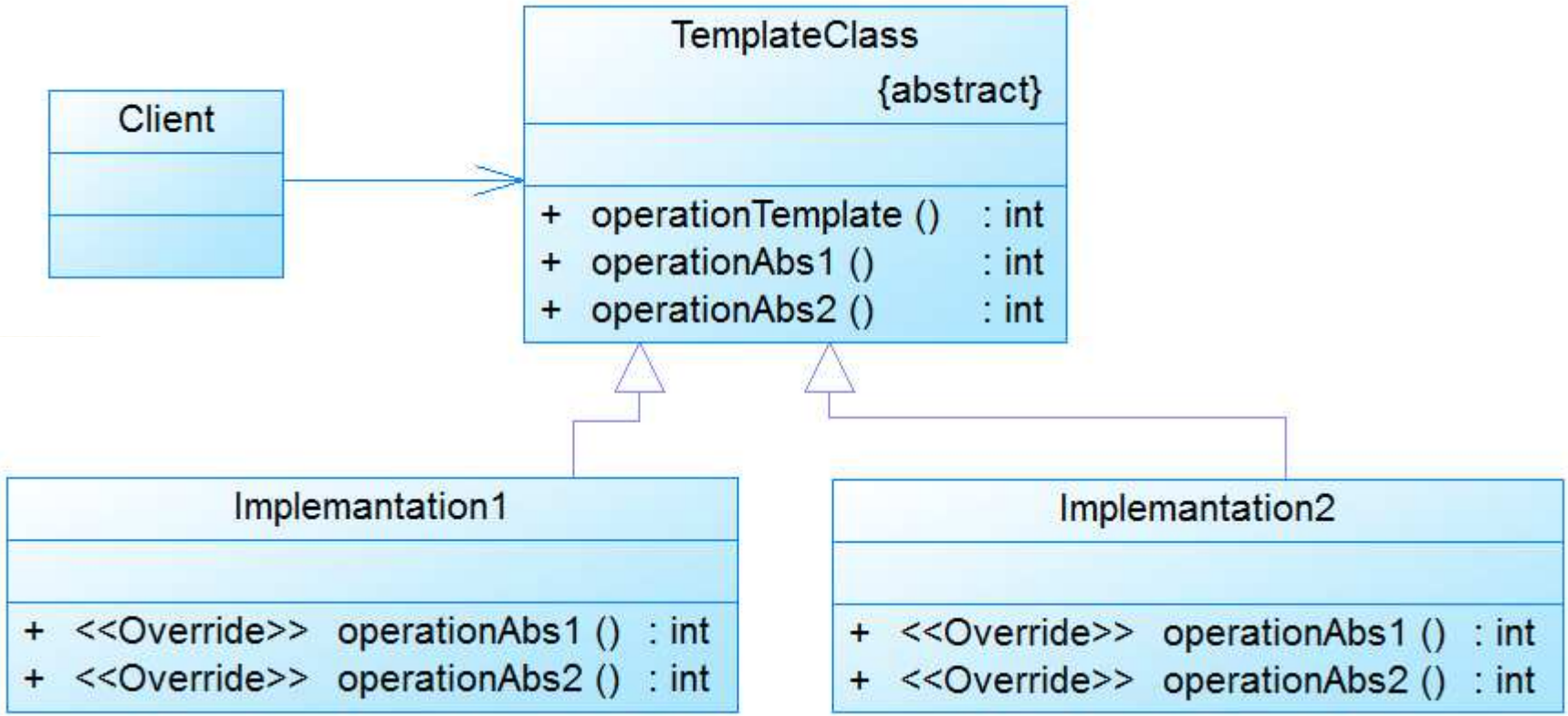
Pattern Template Method

- Catégorie :
  - Comportement
- Objectif du pattern
  - *Définir le squelette d'un algorithme en déléguant certaines étapes à des sous-classes.*
- Résultat :
  - Le Design Pattern permet d'isoler les parties variables d'un algorithme.
- Raisons d'utilisation :
  - Une classe possède un fonctionnement global, mais les détails de son algorithme doivent être spécifiques à ses sous-classes.

Responsabilités

- **TemplateClass**: définit des méthodes abstraites primitives. La classe implémente le squelette d'un algorithme qui appelle les méthodes primitives.
- **Implentation1**, **Implementation2** : sont des sous-classes concrète de TemplateClass. Elle implémente les méthodes utilisées par l'algorithme de la méthode operationTemplate() de TemplateClass.
- **La partie cliente** appelle la méthode de TemplateClass qui définit l'algorithme.

Design pattern Template Method



Implémentation

/* TemplateClass.java */

```
package tm;

public abstract class TemplateClass {
    public int operationTemplate(){
      int a=operationAbs1();
       int somme=0;
       for(int i=0;i<a;i++){
         somme+=operationAbs2();
       }
       return somme;
    }
    protected abstract int operationAbs1();
    protected abstract int operationAbs2();
}
```

Implémentation

/* Implementation1.java */

```
package tm;
public class Implementation1 extends TemplateClass {
@Override
protected int operationAbs1() {
 return 8;
}

@Override
protected int operationAbs2() {
 return 12;
}
}
```

Implémentation

/* Implementation2.java */

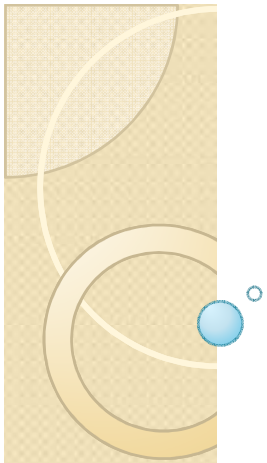
```
package tm;
public class Implementation2  extends TemplateClass{
@Override
protected int operationAbs1() {
  return 34;
}
@Override
protected int operationAbs2() {
 // TODO Auto-generated method stub
  return 56;
}
}
```

Implémentation

/* Implementation2.java */

```
import tm.Implementation1;
import tm.Implementation2;
import tm.TemplateClass;
public class Application {
public static void main(String[] args) {
   TemplateClass t1=new Implementation1();
   System.out.println(t1.operationTemplate());
   t1=new Implementation2();
   System.out.println(t1.operationTemplate());
}
}
```

```
96
1904
```



Pattern Command

Pattern Command

- Catégorie :
  - Comportement
- Objectif du pattern
  - *Encapsuler une requête sous la forme d' objet.*
  - *Paramétrer facilement des requêtes diverses.*
  - *Permettre des opérations réversibles.*
- Résultat :
  - Le Design Pattern permet d'isoler une requête.

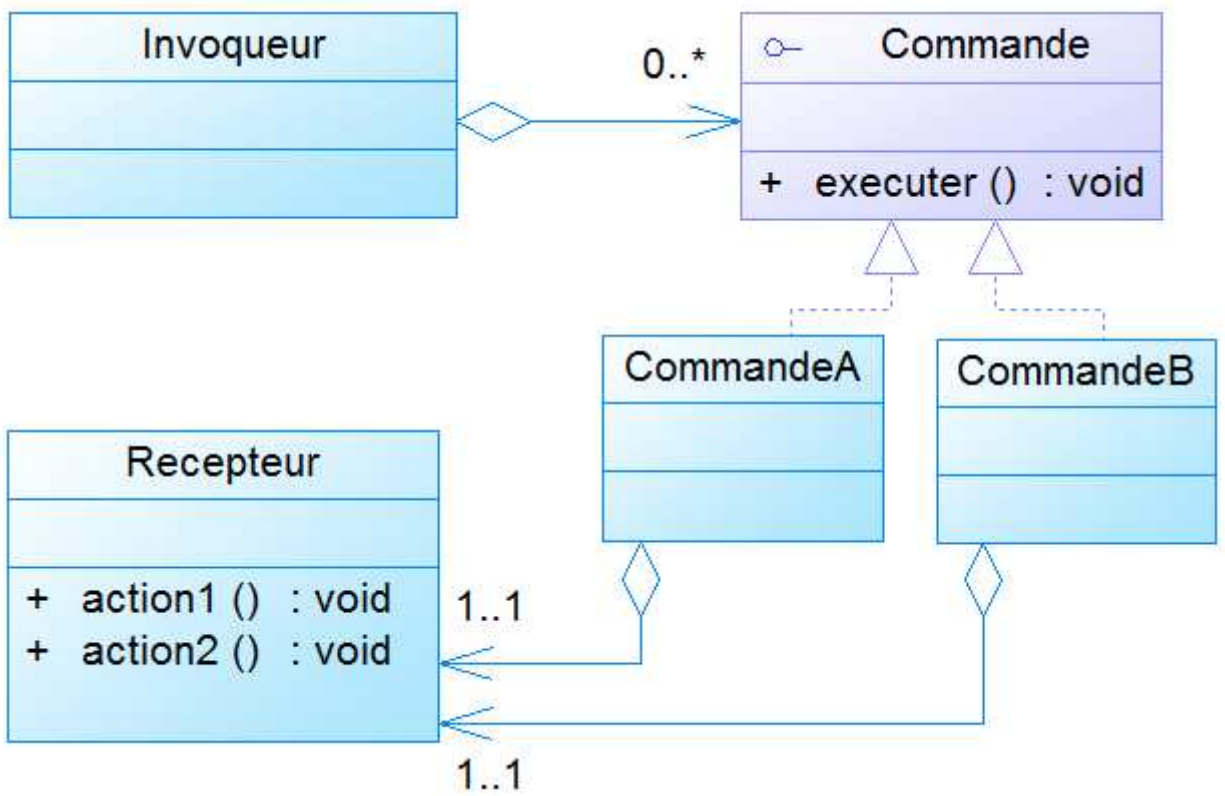
Pattern Command

- Raison d'utilisation :
  - Le système doit traiter des requêtes. Ces requêtes peuvent provenir de plusieurs émetteurs.
  - Plusieurs émetteurs peuvent produire la même requête.
  - Les requêtes doivent pouvoir être annulées.
  - Cela peut être le cas d'une IHM avec des boutons de commande, des raccourcis clavier et des choix de menu aboutissant à la même requête.
  - La requête est encapsulée dans un objet : la commande.
  - Chaque commande possède un objet qui traitera la requête : le récepteur.
  - La commande ne réalise pas le traitement, elle est juste porteuse de la requête.
  - Les émetteurs potentiels de la requête (éléments de l'IHM) sont des invoqueurs.
  - Plusieurs invoqueurs peuvent se partager la même commande.

Responsabilités

- **Commande** : définit l'interface d'une commande.
- **CommandeA** et **CommandeB** : implémentent une commande. Chaque classe implémente la méthode *executer()*, en appelant des méthodes de l'objet **Recepteur**.
- **Invoqueur** : déclenche la commande. Il appelle la méthode *executer()* d'un objet Commande.
- **Recepteur** : reçoit la commande et réalise les opérations associées. Chaque objet Commande concret possède un lien avec un objet Recepteur.
- **La partie cliente** configure le lien entre les objets Commande et le Recepteur.

Design pattern Command



Implémentation

/* Recepteur.java */

```
package cmd;
public class Recepteur {
  public void action1(){
   System.out.println("Action 1 du récepteur");
  }
  public void action2(){
   System.out.println("Action 2 du récepteur");
  }
}
```

Implémentation

/* Commande.java */

```
package cmd;
public interface Commande {
   public void executer();
}
```

/* CommandeA.java */

```
package cmd;
public class CommandeA implements Commande {
  private Recepteur recepteur;
  public CommandeA(Recepteur recepteur) {
    this.recepteur = recepteur;
  }
  @Override
  public void executer() {
    recepteur.action1();
  }
}
```

Implémentation

/* CommandeB.java */

```
package cmd;
public class CommandeB implements Commande {
  private Recepteur recepteur;
  public CommandeB(Recepteur recepteur) {
    this.recepteur = recepteur;
  }
  @Override
  public void executer() {
    recepteur.action2();
  }
}
```

Implémentation

/* Invoqueur.java */

```
package cmd;
import java.util.HashMap;
import java.util.Map;
public class Invoqueur {
   private Map<String, Commande> commades=new HashMap<String,
Commande>();
   public void addCommande(String cmdName,Commande cmd){
     commades.put(cmdName, cmd);
   }
   public void invoquer(String cmdName){
    Commande cmd=commades.get(cmdName);
    if (cmd!=null) cmd.executer();
   }
}
```

Implémentation

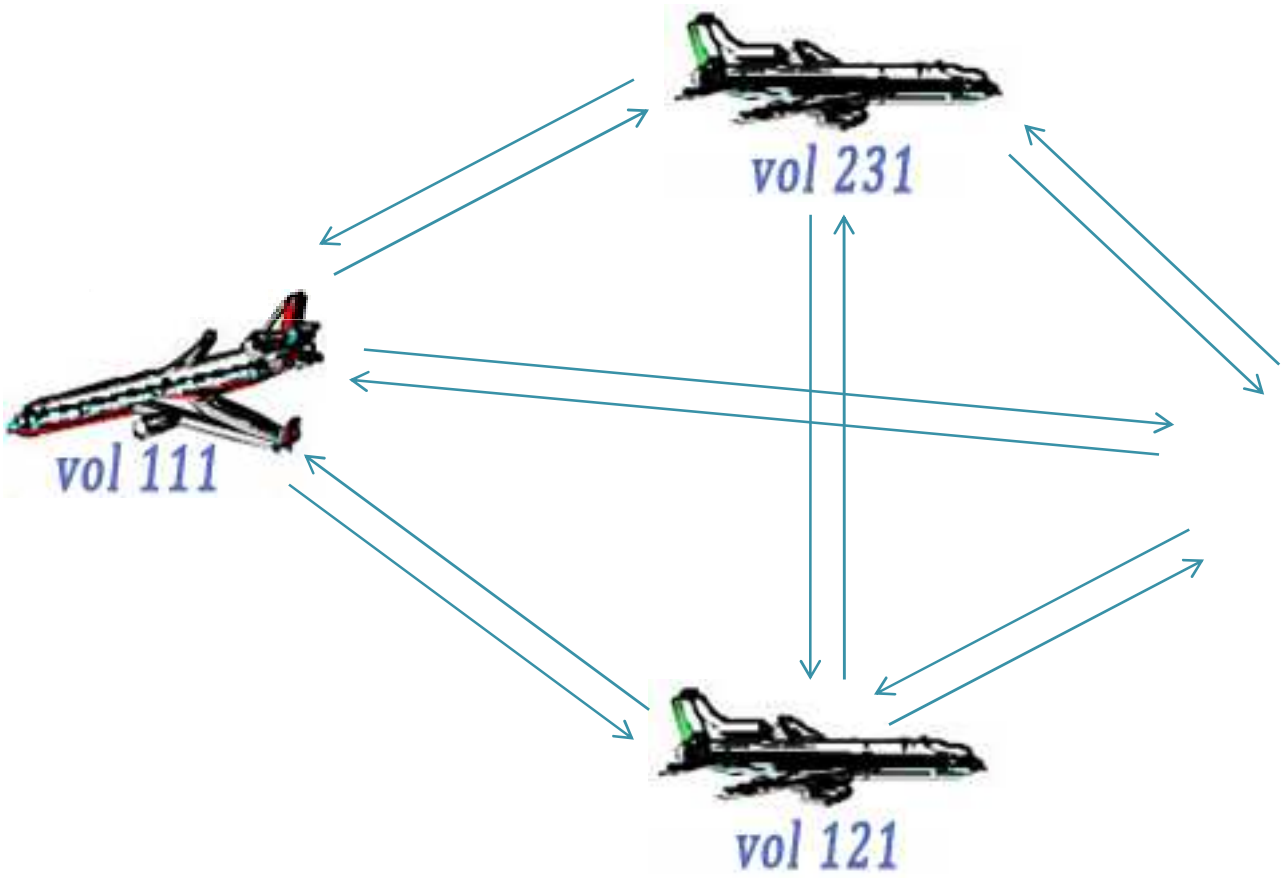
/* Application.java */

```
package cmd;
public class Application {
 public static void main(String[] args) {
   Recepteur rec=new Recepteur();
   CommandeA commandeA=new CommandeA(rec);
   CommandeB commandeB=new CommandeB(rec);

    Invoqueur invoqueur=new Invoqueur();
    invoqueur.addCommande("CA", commandeA);
    invoqueur.addCommande("CB", commandeB);

    invoqueur.invoquer("CA");invoqueur.invoquer("CB");
    invoqueur.invoquer("CC");
 }
}
```

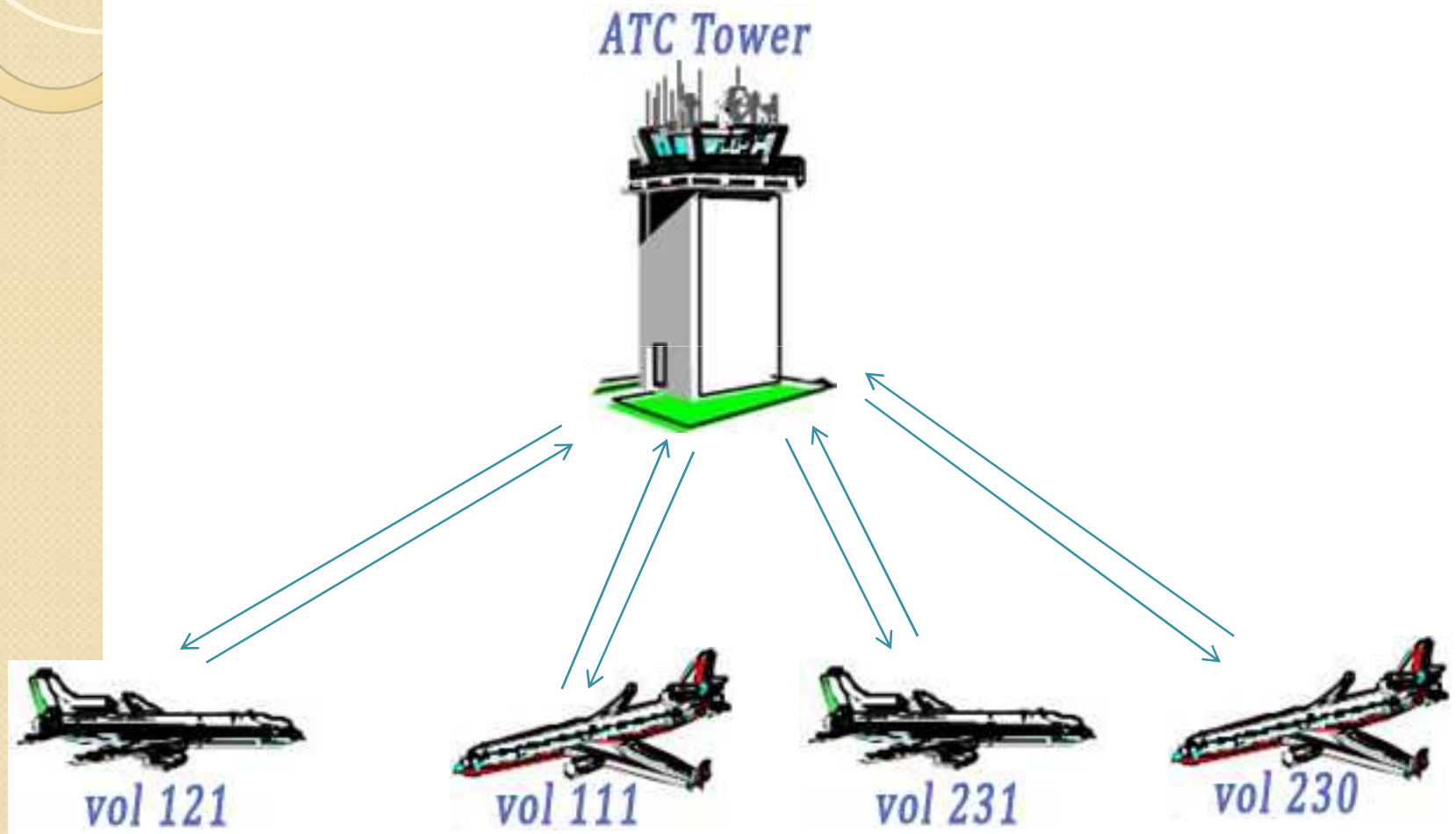
Action 1 du récepteur
Action 2 du récepteur

Pattern Mediator

Exemple (contrôle du trafic aérien)



Solution :



Pattern Mediator

- Catégorie :
  - Comportement
- Objectif du pattern
  - *Gérer la transmission d'informations entre des objets interagissant entre eux.*
  - *Avoir un couplage faible entre les objets puisqu'ils n'ont pas de lien direct entre eux.*
  - *Pouvoir varier leur interaction indépendamment.*
- Résultat :
  - Le Design Pattern permet d'isoler la communication entre des objets.

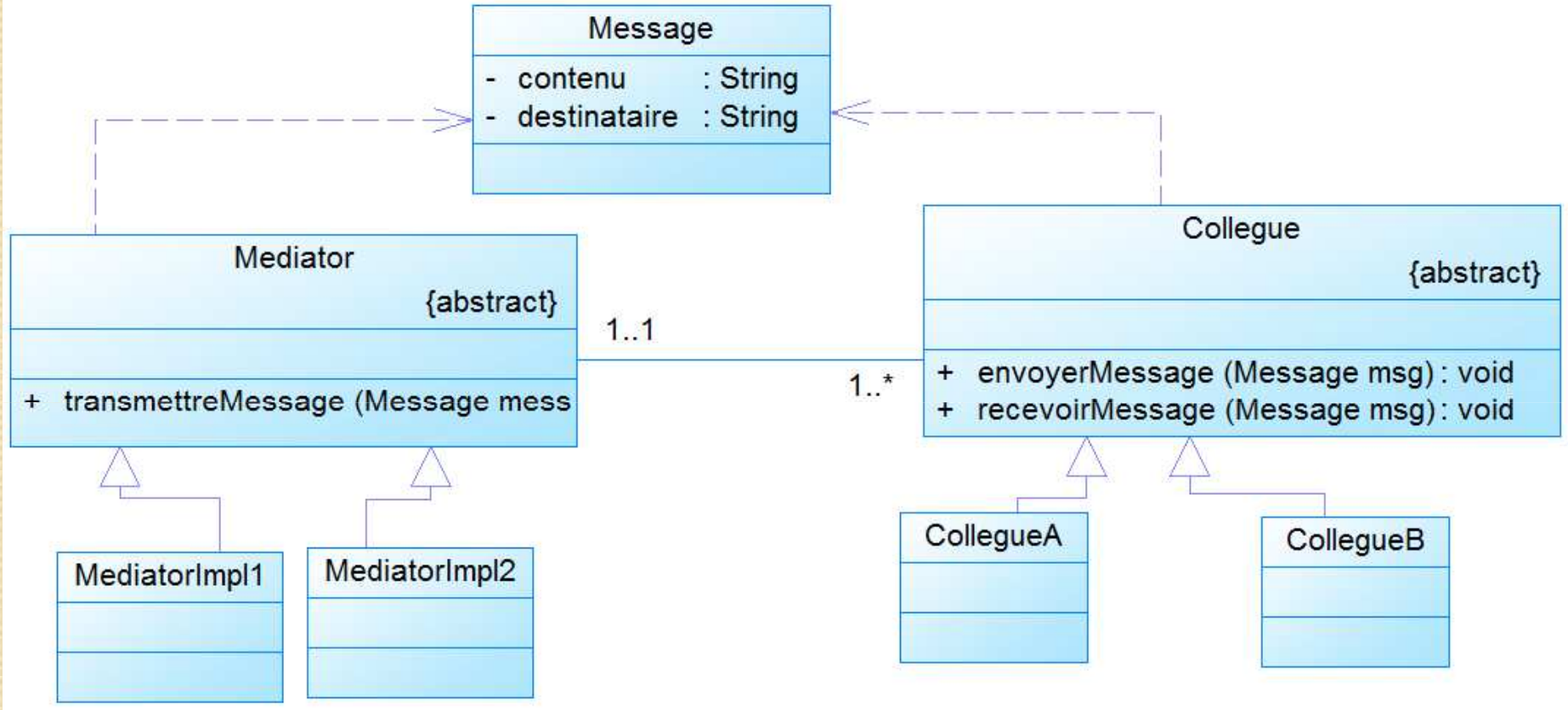
Pattern Mediator

- Raison d'utilisation :
  - Différents objets ont des interactions. Un événement sur l'un provoque une action ou des actions sur un autre ou d'autres objets.
  - Besoin de centraliser le contrôle et les communications complexes entre objets apparentés.
  - Construire un objet dont la vocation est la gestion et le contrôle des interactions complexes entre un ensemble d'objets sans que les éléments doivent se connaître mutuellement.

Responsabilités

- **Collegue** : définit l'interface d'un collègue. Il s'agit d'une famille d'objets qui s'ignorent entre eux mais qui doivent se transmettre des informations.
- **CollegueA** et **CollegueB** : sont des sous-classes concrètes de l'interface Collegue.
- Elles ont une référence sur un objet Mediateur auquel elles transmettront les informations.
- **Mediateur** : définit l'interface de communication entre les objets Collegue.
- **ConcreteMediateur** : implémente la communication et maintient une référence sur les objets Collegue.

Design pattern Mediator



Implémentation

/* Message.java */

```
package med;
public class Message {
  private String message;   private String expediteur;
  private String destinataire;
public Message() { }
public Message(String message, String destinataire) {
  this.message = message;
  this.destinataire = destinataire;
}
@Override
public String toString() {
return "Message [message=" + message + ", expediteur=" + expediteur
+ ", destinataire=" + destinataire + "]";
}
// Getters et Setters
}
```

Implémentation

/* mediateur.java */

```
package med;
import java.util.HashMap;
import java.util.Map;
public abstract class Mediateur {
  protected Map<String, Collegue> collegues=new HashMap<String,
  Collegue>();
  public abstract void transmettreMessage(Message m);

  public void addCollegue(String ref,Collegue a){
    collegues.put(ref, a);
  }
}
```

Implémentation

/* Collegue.java */

```
package med;
public abstract class Collegue {
  protected String name;
  protected Mediateur mediateur;

  public Collegue(String name,Mediateur mediateur) {
  this.name=name;
  this.mediateur = mediateur;
  mediateur.addCollegue(name, this);
  }
  public abstract void envoyerMessage(Message m);
  public abstract void recevoirMessage(Message m);

}
```

Implémentation

/* MediateurImpl.java */

```
package med;
import java.util.ArrayList; import java.util.List;
public class MediateurImpl1 extends Mediateur {
private List<Message> conversations=new ArrayList<Message>();
    @Override
    public void transmettreMessage(Message m) {
     System.out.println("--------- Début Médiateur ----------");
     System.out.println("Enregistrement du message");
     conversations.add(m);
     System.out.println("Transmission du message");
     System.out.println("From :"+m.getExpediteur());
     System.out.println("To:"+m.getDestinataire());
     Collegue destinataire=collegues.get(m.getDestinataire());
     destinataire.recevoirMessage(m);
     System.out.println("--------- Fin Médiateur ----------");
}
public void analyserConversation(){
for(Message m:conversations) System.out.println(m.toString());
}
}
```

Implémentation

/* CollegueA.java */

```
package med;

public class CollegueA extends Collegue {

public CollegueA(String name,Mediateur mediateur) { super(name,mediateur); }

@Override
public void envoyerMessage(Message m) {
  System.out.println("----------------------");
  System.out.println("Collègue nom="+name+", Envoi de message");
  m.setExpediteur(this.name); mediateur.transmettreMessage(m);
  System.out.println("----------------------");
}

@Override
public void recevoirMessage(Message m) {
  System.out.println("----------------------");
  System.out.println("Collègue nom="+name+", Réception du message");
  System.out.println("From :"+m.getExpediteur());
  System.out.println("Contenu:"+m.getMessage());
  System.out.println("Traitement du message par ....."+this.name);
  System.out.println("--------------------");
}
}
```

Implémentation

/* CollegueB.java */

```
package med;

public class CollegueB extends Collegue {

public CollegueB(String name,Mediateur mediateur) { super(name,mediateur); }

@Override
public void envoyerMessage(Message m) {
  System.out.println("----------------------");
  System.out.println("Collègue nom="+name+", Envoi de message");
  m.setExpediteur(this.name); mediateur.transmettreMessage(m);
  System.out.println("----------------------");
}

@Override
public void recevoirMessage(Message m) {
  System.out.println("----------------------");
  System.out.println("Collègue nom="+name+", Réception du message");
  System.out.println("From :"+m.getExpediteur());
  System.out.println("Contenu:"+m.getMessage());
  System.out.println("Traitement du message par ....."+this.name);
  System.out.println("--------------------");
}
}
```

Implémentation

/* Application.java */

```
import med.*;
public class Application {
public static void main(String[] args) {
  MediateurImpl1 mediateur=new MediateurImpl1();
  College a1=new CollegueA("C1",mediateur);
  College a2=new CollegueA("C2",mediateur);
  College b1=new CollegueB("C3",mediateur);
  a1.envoyerMessage(new Message("je suis à 20 m","C2"));
} }
```

```
--------------------
Collègue nom=C1, Envoi de message
--------- Début Médiateur ----------
Enregistrement du message
Transmission du message
From :C1
To:C2
```

```
--------------------
Collègue nom=C2, Réception du message
From :C1
Contenu:je suis à 20 m
Traitement du message par .....C2
------------------
--------- Fin Médiateur ----------
--------------------
```