Design Patterns
Part 7

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

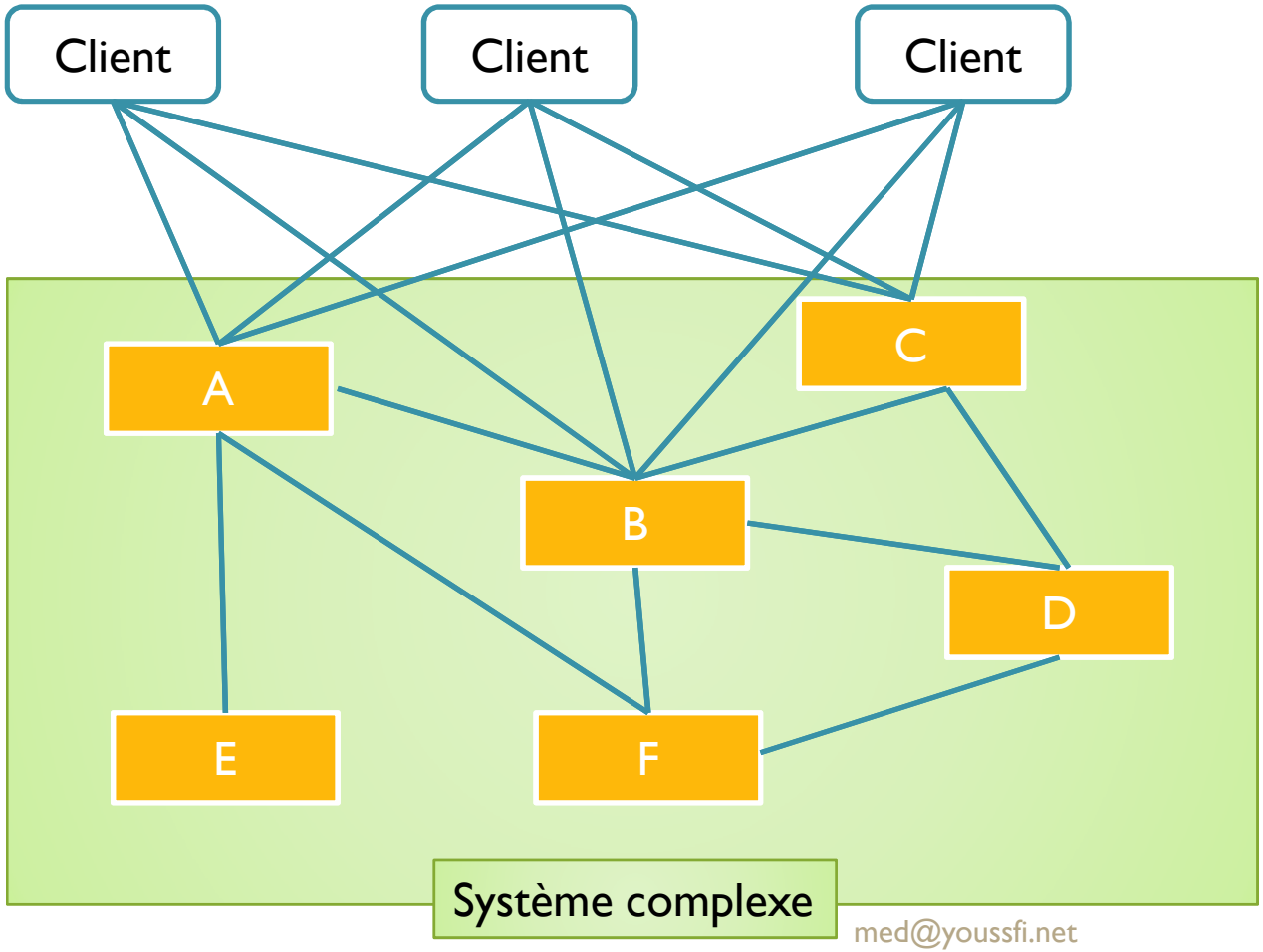
Recherche : http://www.researchgate.net/profile/Youssfi_Mohamed/publications

Patterns :

- Facade
- Bridge
- FlyWeight

Problème : Système complexe

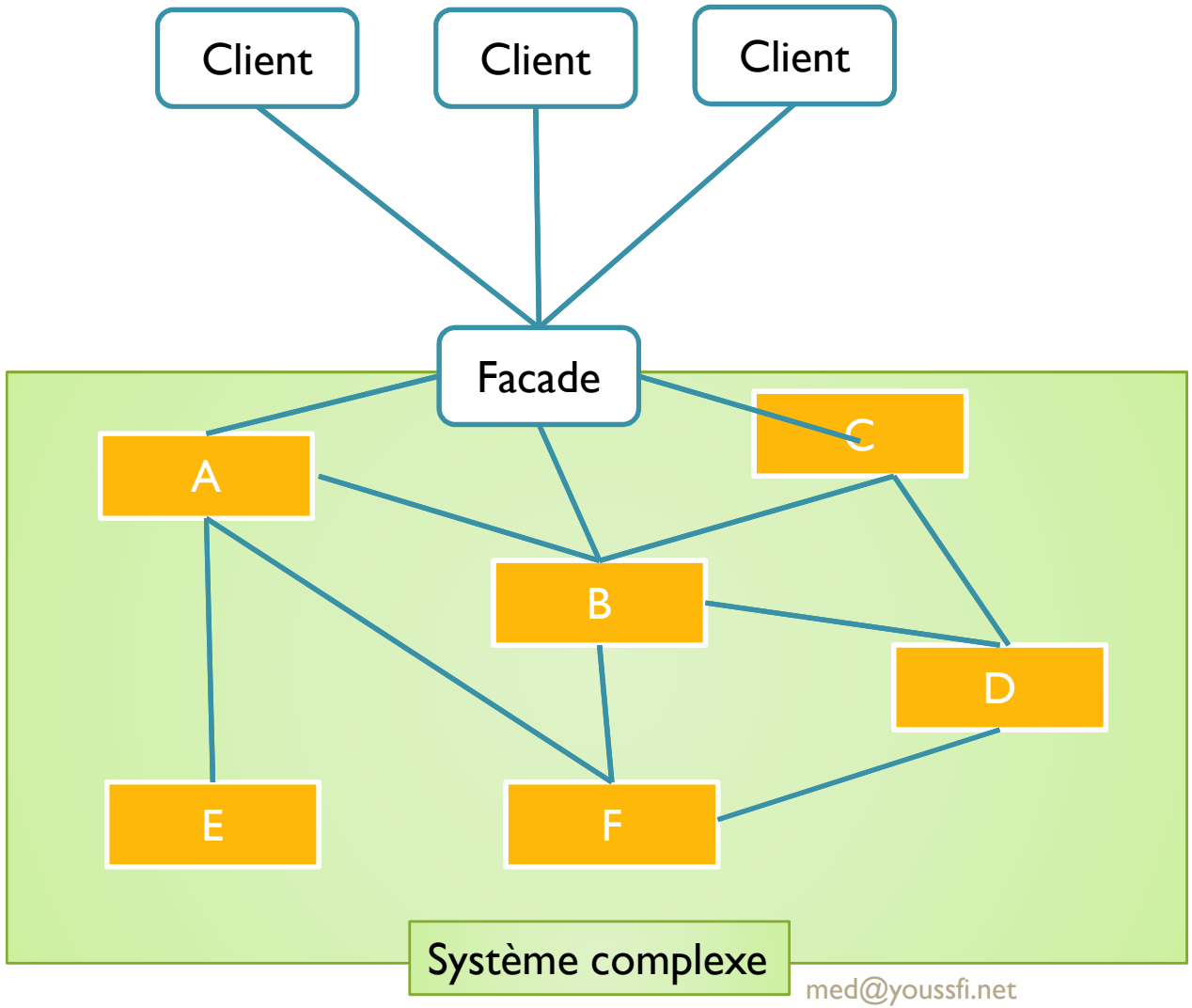
Un sous système est complexe s'il fait appel à plusieurs interfaces



??!!

Pattern Façade

Créer une seule façade pour ce sous-système.

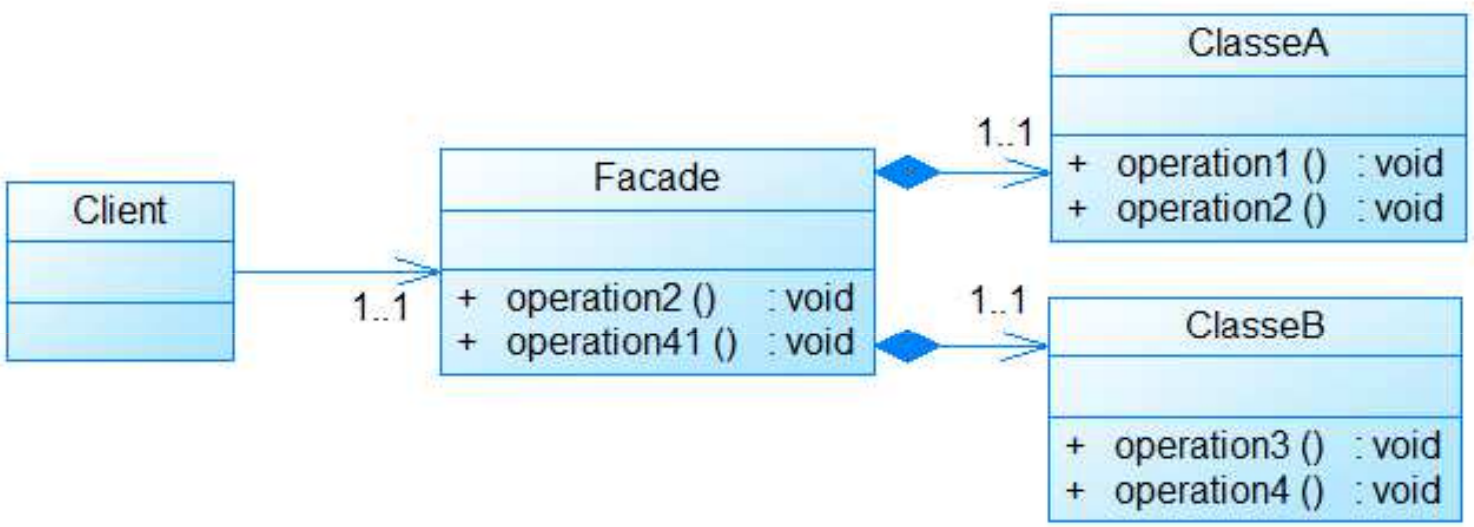


??!!

Pattern Facade

- Catégorie :
  - Structure
- Objectif du pattern
  - *Fournir une interface unique en remplacement d'un ensemble d'interfaces d'un sous-système.*
  - *Définir une interface de haut niveau pour rendre le sous-système plus simple d'utilisation..*
- Résultat :
  - Le Design Pattern permet d'isoler les fonctionnalités d'un sous-système utiles à la partie cliente.

Diagramme de classes du pattern Facade



Raison d'utilisation

- Le système comporte un sous-système complexe avec plusieurs interfaces.
- Certaines de ces interfaces présentent des opérations qui ne sont pas utiles au reste du système.
- Cela peut être le cas d'un sous-système communiquant avec des outils de mesure ou d'un sous-système d'accès à la base de données.
- Il serait plus judicieux de passer par une seule interface présentant seulement les opérations utiles.
- Une classe unique, la façade, présente ces opérations réellement nécessaires.

Responsabilités

- **ClasseA et ClasseB** : implémentent diverses fonctionnalités.
- **Facade** : présente certaines fonctionnalités. Cette classe utilise les implémentations des objets ClasseA et ClasseB. Elle expose une version simplifiée du sous-système ClasseA-ClasseB.
- La **partie cliente** fait appel aux méthodes présentées par l'objet Facade. Il n'y a donc pas de dépendances entre la partie cliente et le sous-système ClasseA-ClasseB.

Implémentation

/* ClasseA.java */

```
public class ClasseA {
  public void operation1(){
   System.out.println("Opération 1 de la classe A");
  }
  public void operation2(){
   System.out.println("Opération 2 de la classe A");
  }
}
```

/* ClasseB.java */

```
public class ClasseB {
  public void operation3(){
   System.out.println("Opération 3 de la classe B");
  }
  public void operation4(){
   System.out.println("Opération 4 de la classe B");
  }
}
```

Implémentation

/* Facade.java */

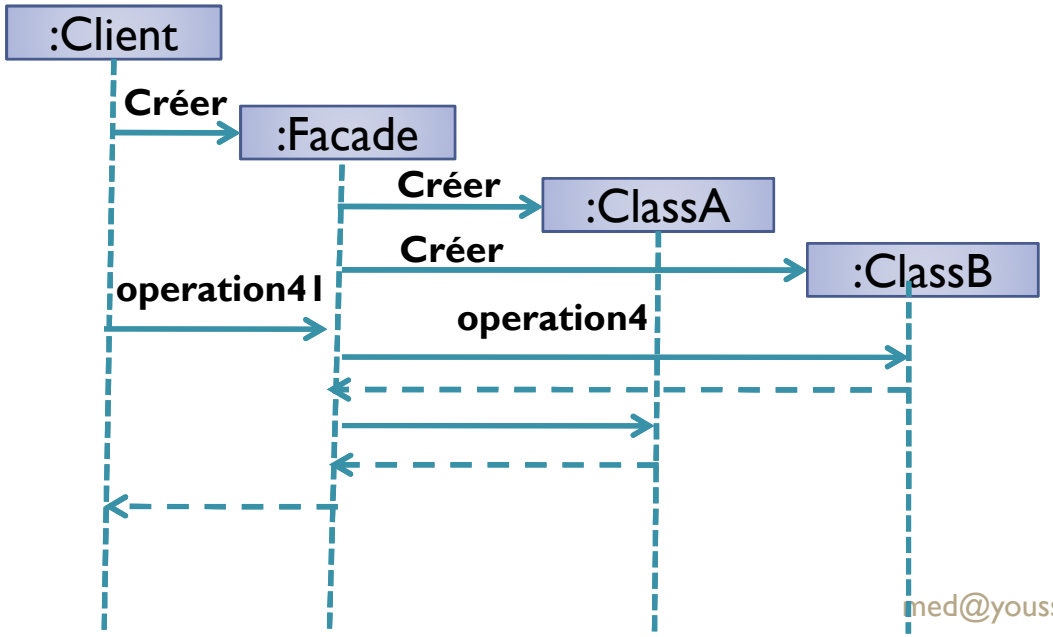
```
public class Facade {
 private ClasseA a=new ClasseA();
 private ClasseB b=new ClasseB();

 public void operation2(){
  System.out.println("Opération 2 de Facade :");
  a.operation2();
}
 public void operation41(){
  System.out.println("Opération 41 de Facade :");
  b.operation4();
  a.operation1();
}
}
```

Implémentation

/* Application.java */

```
public class Application {
 public static void main(String[] args) {
   Facade facade=new Facade();
   facade.operation2();    facade.operation41();
 }
}
```

```
Opération 2 de Facade :
Opération 2 de la classe A
Opération 41 de Facade :
Opération 4 de la classe B
Opération 1 de la classe A
```



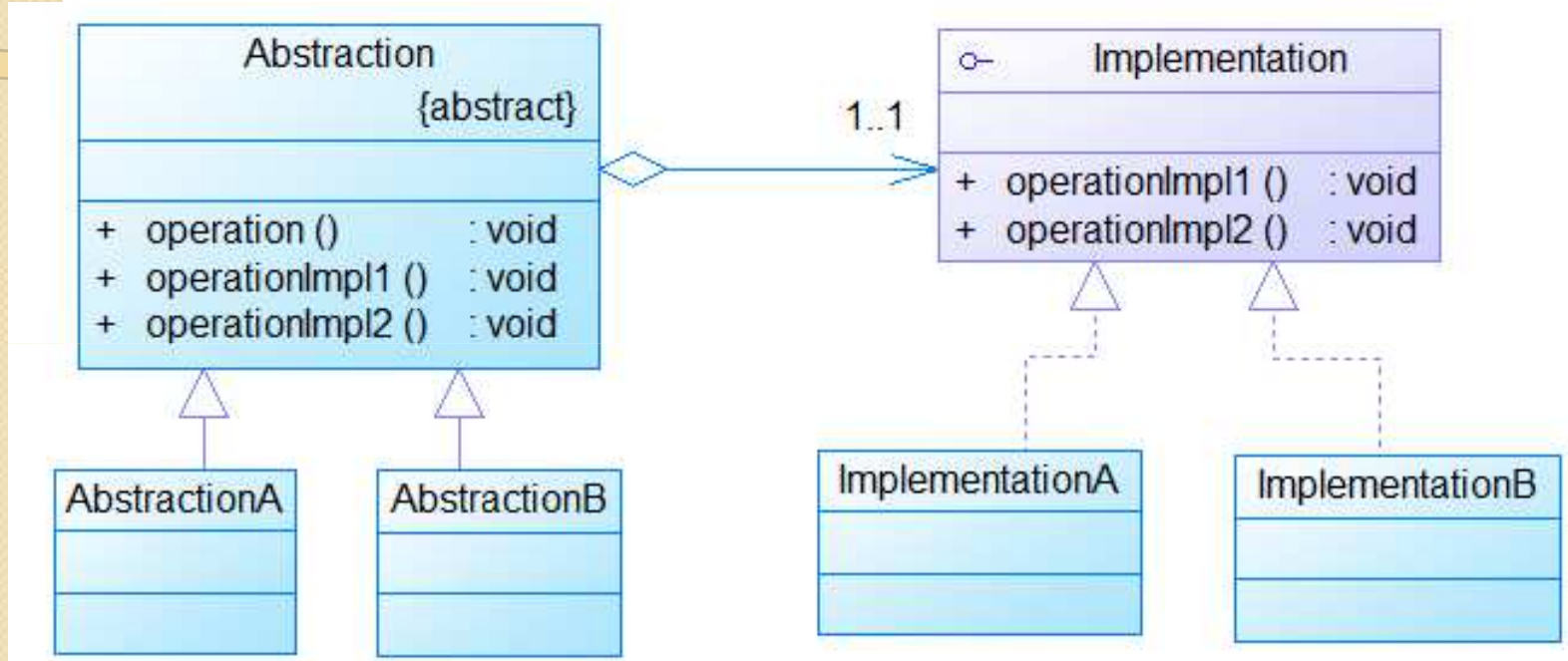
PATTERN BRIDGE

med@youssfi.net

Pattern Bridge

- Catégorie :
  - Structure
- Objectif du pattern
  - *Découpler l'abstraction d'un concept de son implémentation.*
  - *Permettre à l'abstraction et l'implémentation de varier indépendamment.*
- Résultat :
  - Le Design Pattern permet d'isoler le lien entre une couche de haut niveau et celle de bas niveau.

Diagramme de classe



Raison d'utilisation

- Le système comporte une couche bas niveau réalisant l'implémentation et une couche haut niveau réalisant l'abstraction. Il est nécessaire que chaque couche soit indépendante.
- Cela peut être le cas du système d'édition de documents d'une application. Pour l'implémentation, il est possible que l'édition aboutisse à une sortie imprimante, une image sur disque, un document PDF, etc...
- Pour l'abstraction, il est possible qu'il s'agisse d'édition de factures, de rapports de stock, de courriers divers, etc...
- Chaque implémentation présente une interface pour les opérations de bas niveau standard (sortie imprimante)
- Chaque abstraction hérite d'une classe effectuant le lien avec cette interface.
- Ainsi les abstractions peuvent utiliser ce lien pour appeler la couche implémentation pour leurs besoins (imprimer facture).

Responsabilités

- **Implementation** : définit l'interface de l'implémentation. Cette interface n'a pas besoin de correspondre à l'interface de l'Abstraction. L'Implementation peut, par exemple, définir des opérations primitives de bas niveau et l'Abstraction des opérations de haut niveau qui utilisent les opérations de l'Implementation.
- **ImplementationA** et **ImplementationB** : sont des sous-classes concrètes de l'implémentation.
- **Abstraction** : définit l'interface de l'abstraction. Elle possède une référence vers un objet Implementation. C'est elle qui définit le lien entre l'abstraction et l'implémentation. Pour définir ce lien, la classe implémente des méthodes qui appellent des méthodes de l'objet Implementation.
- **AbstractionA** et **AbstractionB** : sont des sous-classes concrètes de l'abstraction. Elle utilise les méthodes définies par la classe Abstraction.
- **La partie cliente** fournit un objet Implementation à l'objet Abstraction. Puis, elle fait appel aux méthodes fournit par l'interface de l'abstraction.

Implémentation

/* ImplementationInterface.java */

```
package impl;
public interface ImplementationInteface {
    public void operationImpl1(String p);
    public void operationImpl2(int p);
}
```

/* ImplementationA.java */

```
package impl;
public class ImplemationA implements ImplementationInteface {
@Override
public void operationImpl1(String p) {
 System.out.println("operationImpl1 de ImplementationA :"+p);
}
@Override
public void operationImpl2(int p) {
 System.out.println("operationImpl2 de ImplementationA :"+p);
}
}
```

Implémentation

/* ImplementationB.java */

```
package impl;
public class ImplemationB implements ImplementationInteface {
@Override
public void operationImpl1(String p) {
 System.out.println("operationImpl1 de ImplementationB :"+p);
}
@Override
public void operationImpl2(int p) {
 System.out.println("operationImpl2 de ImplementationB :"+p);
}
}
```

Implémentation

/* Abstraction.java */

```
package abs;
import impl.ImplementationInteface;
public abstract class Abstraction {
  private ImplementationInteface implementation;

  public void operationImpl1(String p) {
       implementation.operationImpl1(p);
  }
  public void operationImpl2(int p) {
       implementation.operationImpl2(p);
  }
  public Abstraction(ImplementationInteface implementation) {
       this.implementation = implementation;
  }
  public abstract void operation();

}
```

Implémentation

/* AbstractionA.java */

```
package abs;
import impl.ImplementationInteface;

public class AbstractionA  extends Abstraction{
  public AbstractionA(ImplementationInteface implementation) {
        super(implementation);
 }

  @Override
  public void operation() {
    System.out.println("Méthode operation de AbstractionA");
    operationImpl1("X");
    operationImpl2(5);
    operationImpl1("Y");
}
}
```

Implémentation

/* AbstractionB.java */

```
package abs;
import impl.ImplementationInteface;

public class AbstractionB  extends Abstraction{

 public AbstractionB(ImplementationInteface implementation) {
   super(implementation);
 }
 @Override
 public void operation() {
    System.out.println("Méthode operation de AbstractionB");
    operationImpl2(9);
    operationImpl2(4);
    operationImpl1("Z");
 }
}
```

Implémentation

/* Facade.java */

```
public class Facade {
 private ClasseA a=new ClasseA();
 private ClasseB b=new ClasseB();

 public void operation2(){
  System.out.println("Opération 2 de Facade :");
  a.operation2();
}
 public void operation41(){
  System.out.println("Opération 41 de Facade :");
  b.operation4();
  a.operation1();
}
}
```

Implémentation

/* Application.java */

```
import abs.*; import impl.*;
public class Application {
 public static void main(String[] args) {
   ImplementationInteface implA=new ImplemationA();
   ImplementationInteface implB=new ImplemationB();
   Abstraction absAA=new AbstractionA(implA); Abstraction absAB=new AbstractionA(implB);
   Abstraction absBA=new AbstractionB(implA); Abstraction absBB=new AbstractionB(implB);
   System.out.println("--------------------");   absAA.operation();
   System.out.println("--------------------");   absAB.operation();
   System.out.println("--------------------");   absBA.operation();
   System.out.println("--------------------");   absBB.operation();
 } }
```

```
--------------------
Méthode operation de AbstractionA
operationImpl1 de ImplementationA :X
operationImpl2 de ImplementationA :5
operationImpl1 de ImplementationA :Y
--------------------
Méthode operation de AbstractionA
operationImpl1 de ImplementationB :X
operationImpl2 de ImplementationB :5
operationImpl1 de ImplementationB :Y
```

```
--------------------
Méthode operation de AbstractionB
operationImpl2 de ImplementationA :9
operationImpl2 de ImplementationA :4
operationImpl1 de ImplementationA :Z
--------------------
Méthode operation de AbstractionB
operationImpl2 de ImplementationB :9
operationImpl2 de ImplementationB :4
operationImpl1 de ImplementationB :Z
```

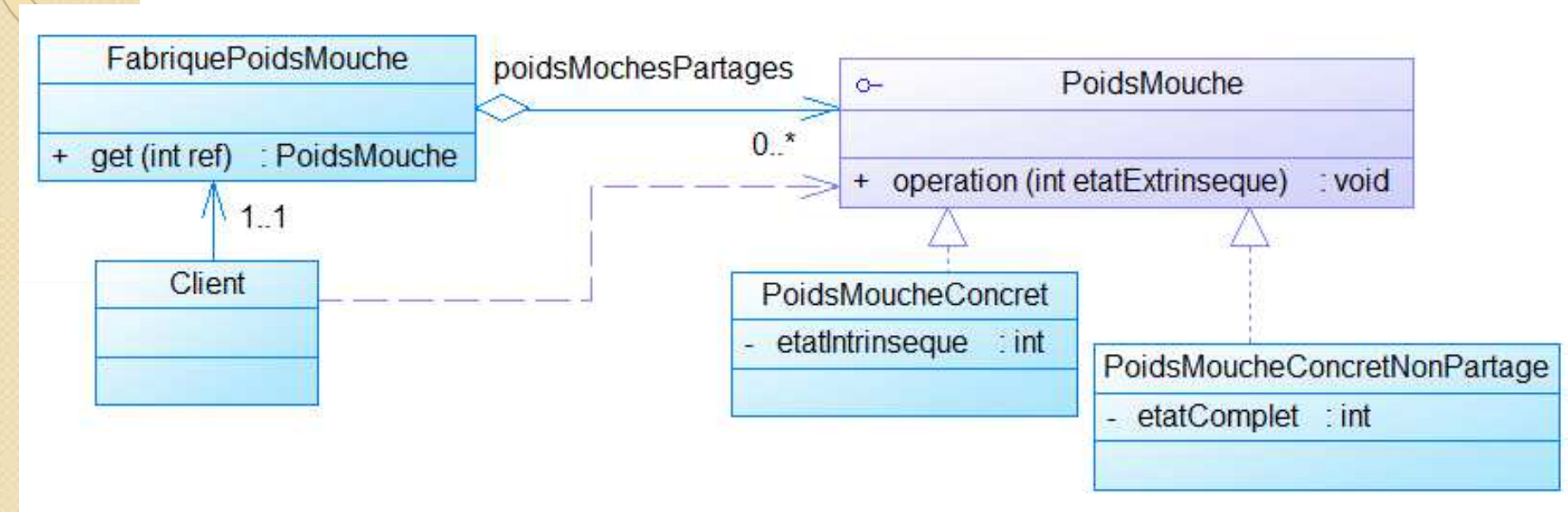
PATTERN FLYWEIGHT POIDS MOUCHE

med@youssfi.net

Pattern FlyWeight

- Catégorie :
  - Structure
- Objectif du pattern
  - *Utiliser le partage pour gérer efficacement un grand nombre d' objets de faible granularité.*
- Résultat :
  - Le Design Pattern permet d'isoler des objets partageables.

Diagramme de classe



Raison d'utilisation

- Un système utilise un grand nombre d' instances. Cette quantité occupe une place très importante en mémoire.
- Or, chacune de ces instances a des attributs extrinsèques (propre au contexte) et intrinsèques (propre à l'objet).
- Cela peut être les caractéristiques des traits dans un logiciel de DAO.
- Le trait a une épaisseur (simple ou double), une continuité (continu, en pointillé), une ombre ou pas, des coordonnées. Les caractéristiques d'épaisseur, de continuité et d'ombre sont des attributs intrinsèques à un trait,
- Tandis que les coordonnées sont des attributs extrinsèques.
- Plusieurs traits possèdent des épaisseurs, continuité et ombre similaires. Ces similitudes correspondent à des styles de trait.

Raison d'utilisation

- En externalisant les attributs intrinsèques des objets (style de trait), on peut avoir en mémoire une seule instance correspondant à un groupe de valeurs (simple-continu-sans ombre, double-pointillé-ombre).
- Chaque objet avec des attributs extrinsèques (trait avec les coordonnées) possède une référence vers une instance d'attributs intrinsèques (style de trait).
- On obtient deux types de poids-mouche : les poids-mouche partagés (style de trait) et les poids mouche non partagés (le trait avec ses coordonnées).
- La partie cliente demande le poids-mouche qui l'intéresse à la fabrique de poids-mouche. S'il s'agit d'un poids mouche non partagé, la fabrique le créera et le retournera.
- S'il s'agit d'un poids-mouche partagé, la fabrique vérifiera si une instance existe. Si une instance existe, la fabrique la retourne, sinon la fabrique la crée et la retourne.

Responsabilités

- **PoidsMouche** : déclare l'interface permettant à l'objet de recevoir et d'agir en fonction de données extrinsèques. On externalise les données extrinsèques d'un objet PoidsMouche afin qu'il puisse être réutilisé.
- **ConcretePoidsMouche** : implémente l'interface poids-mouche. Les informations contenues dans un ConcretePoidsMouche sont intrinsèques (sans lien avec son contexte). Puisque, ce type de poids mouche est obligatoirement partagé.
- **ConcretePoidsMoucheNonPartage** : implémente l'interface poids-mouche. Ce type de poids-mouche n'est pas partagé. Il possède des données intrinsèques et extrinsèques.
- **PoidsMoucheFabrique** : fournit une méthode retournant une instance de PoidsMouche. Si les paramètres de l'instance souhaitée correspondent à un PoidsMouche partagé, l'objet PoidsMoucheFabrique retourne une instance déjà existante. Sinon l'objet PoidsMoucheFabrique crée une nouvelle instance.
- **La partie cliente** demande à PoidsMoucheFabrique de lui fournir un PoidsMouche.

Implémentation

/* PoidsMouche.java */

```
public interface PoidsMouche {
  public void afficher(String context);
}
```

/* ImplementationA.java */

```
public class PoidsMoucheConcret implements PoidsMouche {
 private String valeur;

 public PoidsMoucheConcret(String valeur) {
  this.valeur = valeur;
 }
 @Override
 public void afficher(String context) {
  System.out.println("Poids mouche avec la valeur "+valeur+"
,context="+context);
}
}
```

Implémentation

/* PoidsMoucheNonPartage.java */

```
public class PoidsMoucheNonPartage implements PoidsMouche {
  private String valeur1;
  private String valeur2;

  public PoidsMoucheNonPartage(String valeur1, String valeur2) {
       this.valeur1 = valeur1;
       this.valeur2 = valeur2;
  }
  @Override
  public void afficher(String context) {
       System.out.println("Poids mouche avec valeut1="+valeur1+"
,valeur2="+valeur2);
  }
}
```

Implémentation

/* FabriquePoidsMouche.java */

```
import java.util.HashMap;
import java.util.Map;
public class FabriquePoidsMouche {
  private Map<String, PoidsMouche> poidsMouchesPartages=
  new HashMap<String,PoidsMouche>();

  public FabriquePoidsMouche() {
   poidsMouchesPartages.put("je",new PoidsMoucheConcret("je"));
   poidsMouchesPartages.put("suis",new PoidsMoucheConcret("suis"));
   poidsMouchesPartages.put("poids",new PoidsMoucheConcret("poids"));
   poidsMouchesPartages.put("mouche",new PoidsMoucheConcret("mouche"));
  }
  public PoidsMouche getPoidsMouche(String val){
    PoidsMouche pm=poidsMouchesPartages.get(val);
    if (pm==null){
      pm=new PoidsMoucheConcret(val);
      poidsMouchesPartages.put(val, pm);
    }
    return pm;
  }
  public PoidsMouche getPoidsMouche(String val1,String val2){
    return new PoidsMoucheNonPartage(val1, val2);
  }
}
```

Implémentation

/* Client.java */

```
public class Client {
public static void main(String[] args) {
    FabriquePoidsMouche fabrique=new FabriquePoidsMouche();
    PoidsMouche pm=fabrique.getPoidsMouche("poids");
    PoidsMouche pmBis=fabrique.getPoidsMouche("poids");
    System.out.println("------------------");pm.afficher("Context1"); pmBis.afficher("Context2");
    System.out.println(pm==pmBis); System.out.println("------------------");
    PoidsMouche pm2=fabrique.getPoidsMouche("flyweight");
    PoidsMouche pm2Bis=fabrique.getPoidsMouche("flyweight");
    pm2.afficher("Context1");    pm2Bis.afficher("Context2");
    System.out.println(pm2==pm2Bis); System.out.println("------------------");
    PoidsMouche pm3=fabrique.getPoidsMouche("A","B");
    PoidsMouche pm3Bis=fabrique.getPoidsMouche("A","B");
    pm3.afficher(null); pm3Bis.afficher(null);
    System.out.println(pm3==pm3Bis); System.out.println("------------------");
}
}
```

```
------------------
Poids mouche avec la valeur poids ,context=Context1
Poids mouche avec la valeur poids ,context=Context2
true
------------------
Poids mouche avec la valeur flyweight ,context=Context1
Poids mouche avec la valeur flyweight ,context=Context2
true
```

```
------------------
Poids mouche avec valeut1=A ,valeur2=B
Poids mouche avec valeut1=A ,valeur2=B
false
------------------
```