# Plan

- Rappel bases OO
- Premiers Principes OO
- Grands principes OO
- Divers
  - Approches de développement
  - Pratiques de programmation
- **Design Patterns OO**
- Restes

# C'est quoi ces fameux *design patterns* ?

- Un patron de conception est un arrangement caractéristique de modules, reconnu comme **bonne pratique** en réponse à un **problème** de conception d'un logiciel
  - Il est issu de l'expérience des concepteurs de logiciels
- Il décrit une **solution standard**, utilisable dans la conception de différents logiciels
  - Il est souvent décrit sous forme de diagrammes comme un arrangement récurrent de **rôles** et **d'actions** joués par des modules d'un logiciel
  - le nom du patron sert de **vocabulaire commun** entre le concepteur et le programmeur
- Il décrit les grandes lignes d'une solution, qui peuvent ensuite être **modifiées** et **adaptées** en fonction des besoins

# Historique

- Formalisés dans le livre du « **Gang of Four** » en 1995
  - **GoF** = Erich Gamma + Richard Helm, Ralph Johnson et John Vlissides
  - intitulé *Design Patterns – Elements of Reusable Object-Oriented Software*
- Inspirés des travaux de l'architecte en bâtiments Christopher Alexander dans les années 70
  - dont son livre *A Pattern Language* définissant un ensemble de patrons d'architecture

# Formalisme des patrons de conception

- Nom
- Description du problème à résoudre
- Description de la solution
  - les éléments de la solution, avec leurs relations
  - souvent modélisée avec une notation orientée objet (**UML** en majorité)
- Conséquences
  - résultats issus de la solution

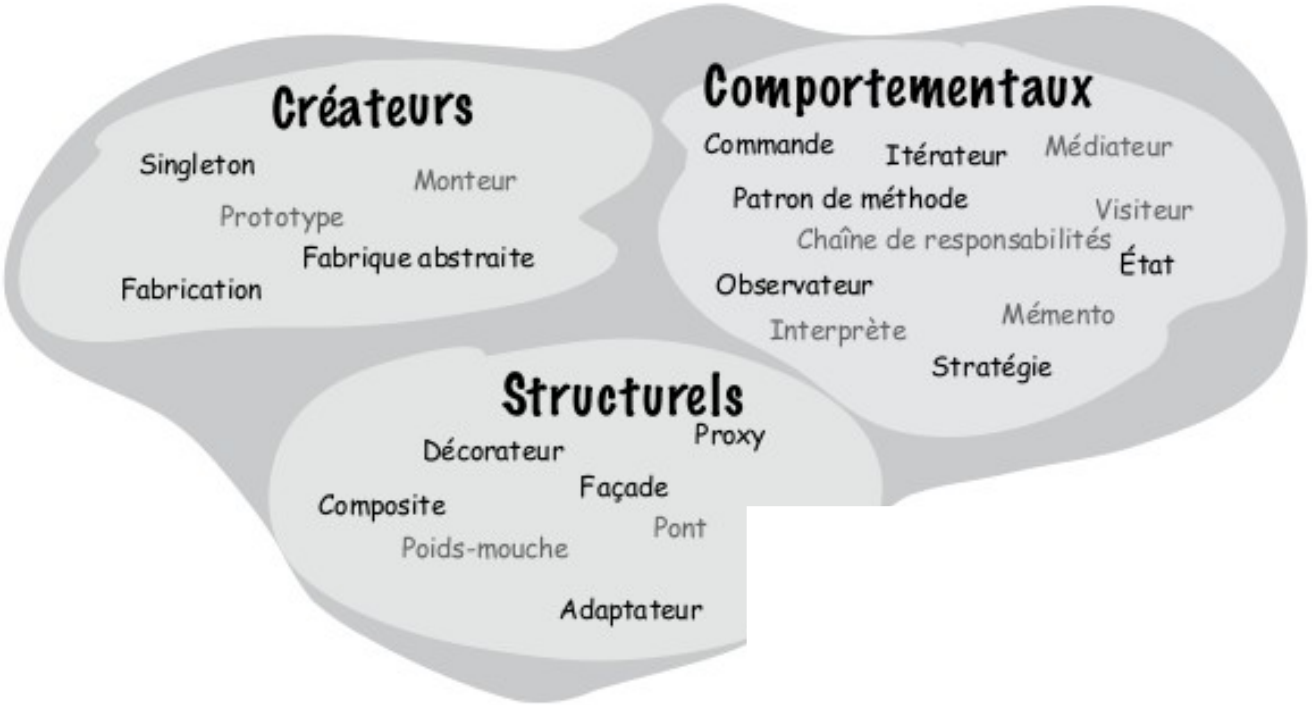
# D'autres patrons ?

- Il existe **23** patrons de conception
- Le **MVC** (Modèle-Vue-Contrôleur) n'est pas l'un d'entre eux
  - C'est une combinaison de 3 patrons de conception
  - Il entre dans la catégorie des patrons d'architecture
- Il existe
  - Les patrons GRASP (*General Responsibility Assignment Software Patterns*) de Craig Larman
  - Les patrons d'architecture
  - Les patrons d'analyse
  - Les patrons spécifiques à une technologie/plateforme
    - Exemple : patrons pour JavaEE (http://www.corej2eepatterns.com/)

# Organisation des patrons

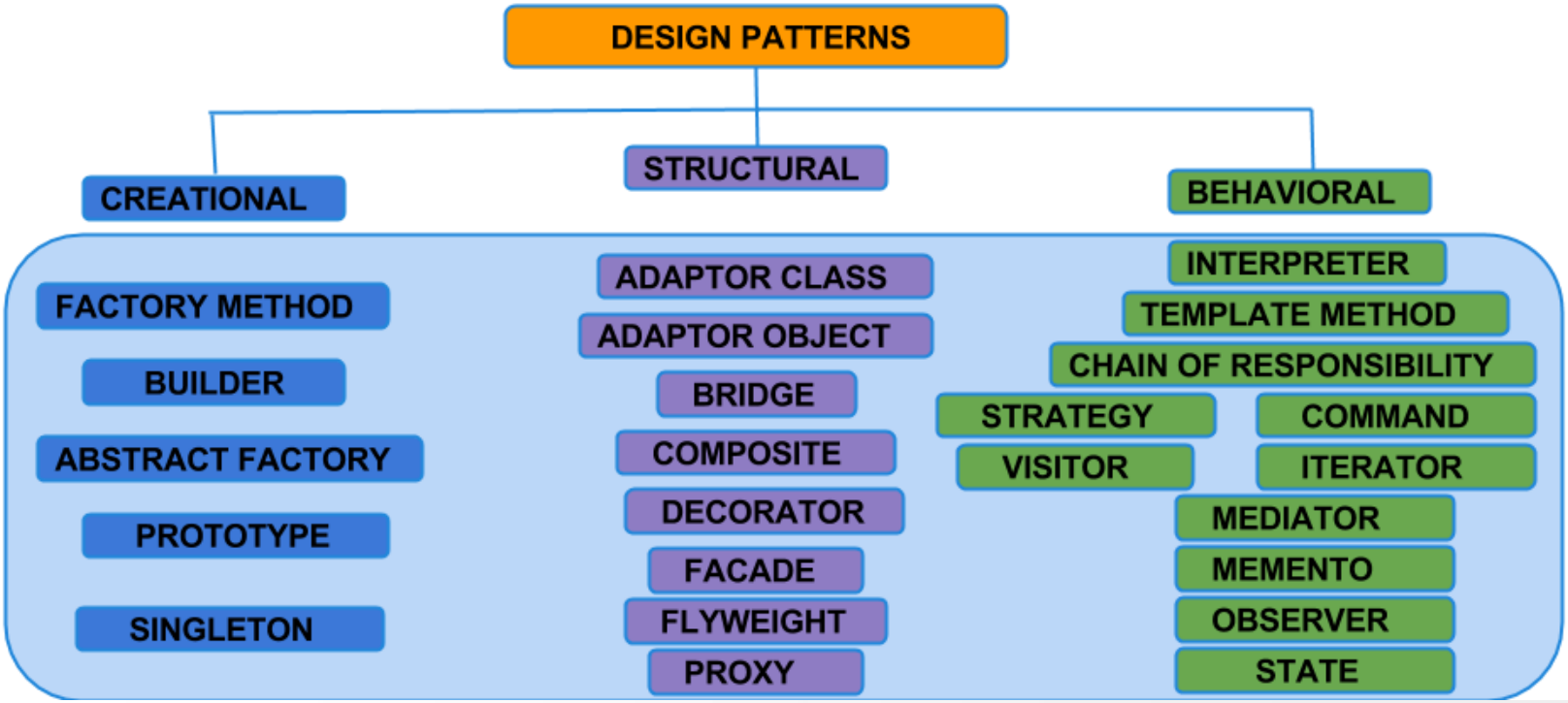
**Les Patterns Créateurs** concernent l'instanciation des objets et fournissent tous un moyen de découpler un client des objets qu'il a besoin d'instancier.

Tout pattern qui est un **Pattern Comportemental** est en rapport avec les interactions des classes et des objets et la distribution des responsabilités.



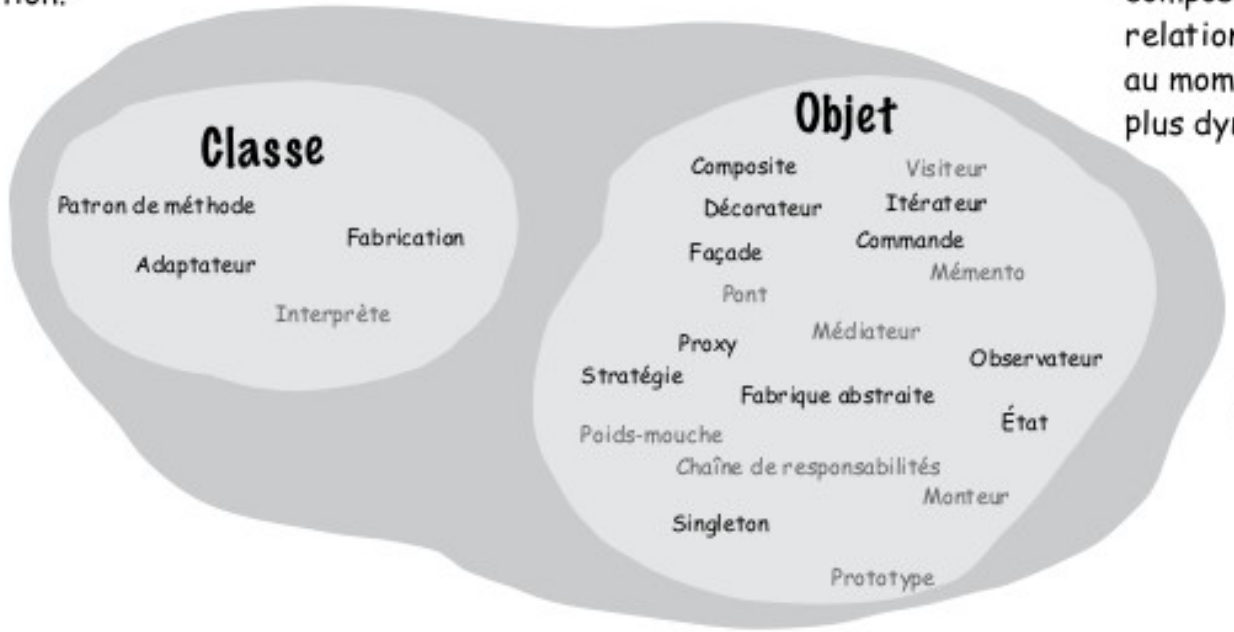

**Les Patterns Structurels** permettent de composer des classes ou des objets pour former des structures plus vastes.

# Les noms anglais...



# Autre critère de classification

**Les patterns de classes** définissent des relations entre classes et sont définis via l'héritage. Dans ces patterns, les relations sont établies au moment de la compilation.




**Les patterns d'objets** décrivent des relations entre objets et sont principalement définis par composition. Dans ces patterns, les relations sont généralement définies au moment de l'exécution et sont plus dynamiques et plus souples.

Notez que les patterns d'objets sont beaucoup plus nombreux que les patterns de classes

# Les patrons que l'on approfondira en TD/TP

- **Fabrique abstraite (Abstract Factory)**
- *Monteur (Builder)*
- *Fabrique (Factory Method)*
- *Prototype (Prototype)*
- *Singleton (Singleton)*
- **Adaptateur (Adapter)**
- *Pont (Bridge)*
- **Objet composite (Composite)**
- **Décorateur (Decorator)**
- *Façade (Facade)*
- *Poids-mouche ou poids-plume (Flyweight)*
- *Proxy (Proxy)*
- *Chaîne de responsabilité (Chain of responsibility)*
- *Commande (Command)*
- *Interpréteur (Interpreter)*
- *Itérateur (Iterator)*
- *Médiateur (Mediator)*
- *Mémento (Memento)*
- **Observateur (Observer)**
- **État (State)**
- **Stratégie (Strategy)**
- **Patron de méthode (Template Method)**
- *Visiteur (Visitor)*

# Les patrons GoF créateurs

# Singleton – introduction orientée codage

- Comment créeriez-vous un seul objet d'une classe donné ?
- Tant que nous avons une classe, pouvons-nous toujours l'instancier une ou plusieurs fois ?
- Sinon ?

**DEMO**

# Singleton - codage

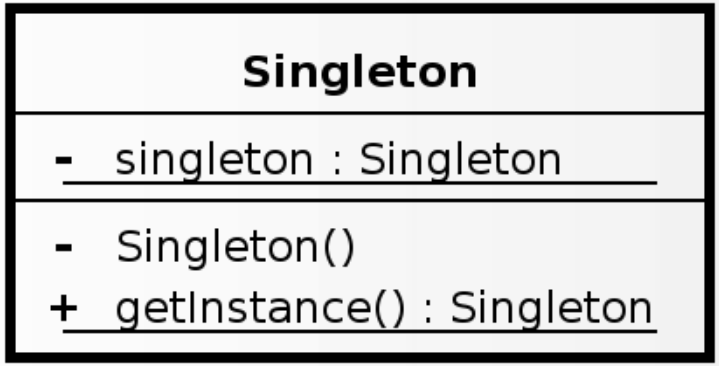
- Premier codage en Java

```
public class Singleton {
    private static Singleton uniqueInstance;
    // autres variables d'instance

    private Singleton() {}

    public static Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }
    // autres méthodes
}
```

Cette version n'est pas thread-safe !
=> il en existe une autre version

# Singleton - définition

- Définition
  - Le Pattern Singleton garantit qu'une classe n'a qu'une seule instance et fournit un point d'accès global à cette instance
- Modélisation de la solution




- Implantations connues dans le *core* de Java
  - `java.lang.Runtime#getRuntime()`
  - `java.awt.Desktop#getDesktop()`

# Application du singleton

- Avant


- Après application sur la classe A ?

# Patron Prototype

- Problème/contexte
  - utilisé lorsque la création d'une instance est complexe ou consommatrice en temps
- Solution
  - **Copie de la première instance mais avec les mêmes propriétés**
  - Le client de cette classe n'invoquant pas l'opérateur "new" mais appelle la méthode **clone()** sur le prototype
- Implantations connues dans le *core* de Java
  - `java.lang.Object#clone()` (the class has to implement `java.lang.Cloneable`)

# Prototype - Exemple de codage en Java

- La classe utilisatrice

```
public class CookieMachine
{
    private Cookie cookie; // peut aussi etre declare : private Cloneable cookie;

    public CookieMachine(Cookie cookie)
    {
        this.cookie = cookie;
    }

    public Cookie makeCookie()
    {
        return (Cookie) cookie.clone();
    }

    public static void main(String args[])
    {
        Cookie        tempCookie =  null;
        Cookie        prot       = new CoconutCookie();
        CookieMachine cm         = new CookieMachine(prot);

        for (int i=0; i<100; i++)
            tempCookie = cm.makeCookie();
    }
}
```

# Prototype - Exemple de codage en Java

- La classe prototype

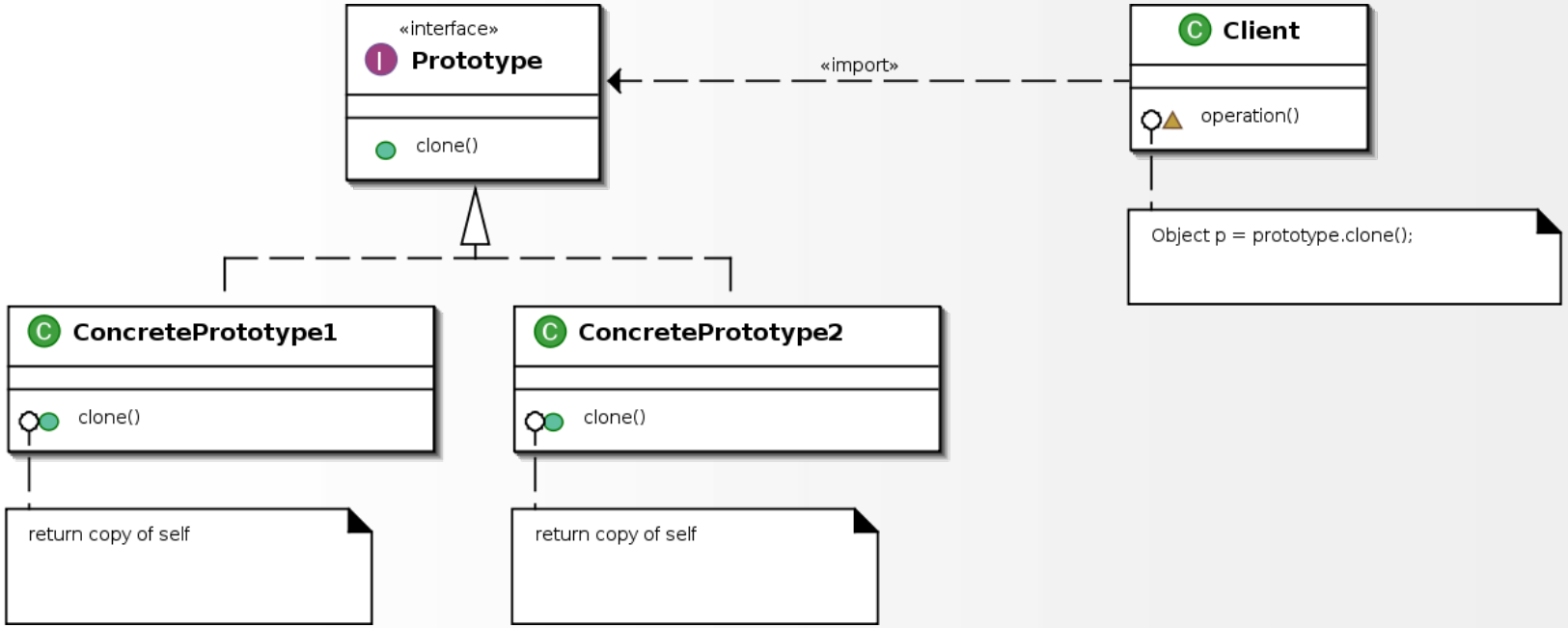
```
public class Cookie implements Cloneable
{
    public Cookie clone()
    {
        try {
            Cookie copy = (Cookie)super.clone();
            // Dans une implémentation réelle de ce patron de
conception, il faudrait
            // créer la copie en dupliquant les objets contenus et en
attribuants des
            // valeurs valides (exemple : un nouvel identificateur
unique pour la copie).
            return copy;
        } catch (CloneNotSupportedException e) {
            return null;
        }
    }
}
```

- Les prototypes concrets à créer

```
public class CoconutCookie extends Cookie { ... }
```

# Prototype

- Modélisation UML



# Patron Monteur

- Problème/contexte
  - création d'une variété d'objets complexes à partir d'un objet source
  - l'objet source peut consister en une variété de parties contribuant individuellement à la création de chaque objet complet
- Solution
  - **encapsuler la création de l'objet complexe dans un autre objet (le monteur) et imposer au client de demandera au monteur de construire la structure de l'objet à sa place**
- Implantations connues dans le *core* de Java
  - `java.lang.StringBuilder#append()`

# Monteur - Exemple de codage en Java

- Les classes clientes

```
class Serveur {
  private MonteurPizza monteurPizza;

  public void setMonteurPizza(MonteurPizza mp) { monteurPizza = mp; }
  public Pizza getPizza() { return monteurPizza.getPizza(); }

  public void construirePizza() {
    monteurPizza.creerNouvellePizza();
    monteurPizza.monterPate();
    monteurPizza.monterSauce();
    monteurPizza.monterGarniture();
  }
}
class ExempleMonteur {
  public static void main(String[] args) {
    Serveur serveur = new Serveur();
    MonteurPizza monteurPizzaHawaii  = new MonteurPizzaHawaii();
    MonteurPizza monteurPizzaPiquante = new MonteurPizzaPiquante();

    serveur.setMonteurPizza(monteurPizzaHawaii);
    serveur.construirePizza();

    Pizza pizzas = serveur.getPizza();
  }
}
```

# Monteur - Exemple de codage en Java

- La classe produit

```
class Pizza {
  private String pate = "";
  private String sauce = "";
  private String garniture = "";

  public void setPate(String pate)          { this.pate = pate; }
  public void setSauce(String sauce)         { this.sauce = sauce; }
  public void setGarniture(String garniture) { this.garniture = garniture; }
}
```

# Monteur - Exemple de codage en Java

- La classe Monteur abstraite

```
abstract class MonteurPizza {
  protected Pizza pizza;

  public Pizza getPizza() { return pizza; }
  public void creerNouvellePizza() { pizza = new Pizza(); }

  public abstract void monterPate();
  public abstract void monterSauce();
  public abstract void monterGarniture();
}
```

- Les classes Monteur concrètes

```
class MonteurPizzaHawaii extends MonteurPizza {
  public void monterPate()      { pizza.setPate("croisée"); }
  public void monterSauce()     { pizza.setSauce("douce"); }
  public void monterGarniture() { pizza.setGarniture("jambon+ananas"); }
}
class MonteurPizzaPiquante extends MonteurPizza {
  public void monterPate()      { pizza.setPate("feuilletée"); }
  public void monterSauce()     { pizza.setSauce("piquante"); }
  public void monterGarniture() { pizza.setGarniture("pepperoni+salami"); }
}
```

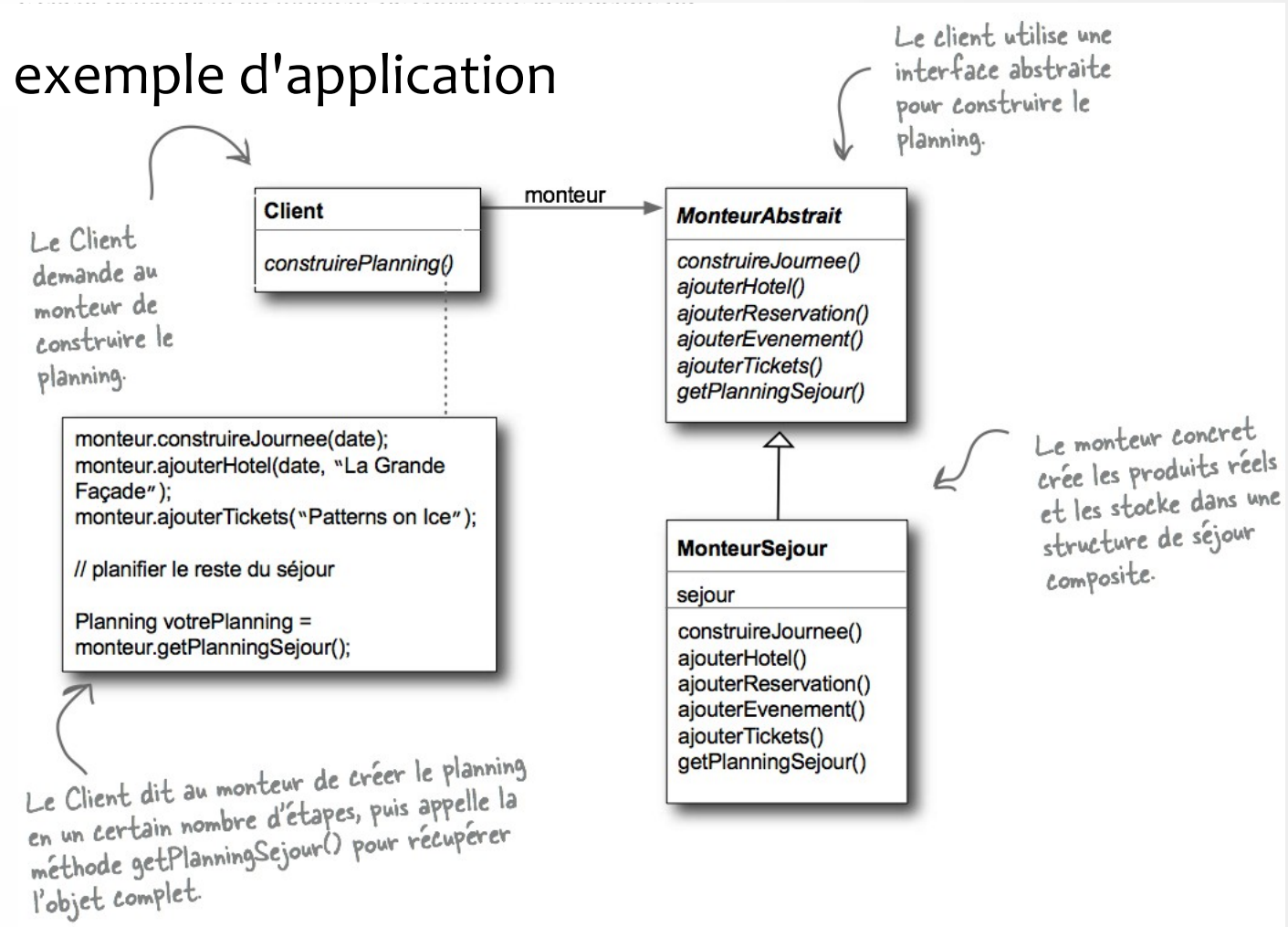
# Monteur - Exemple de codage en Java

- La classe produit

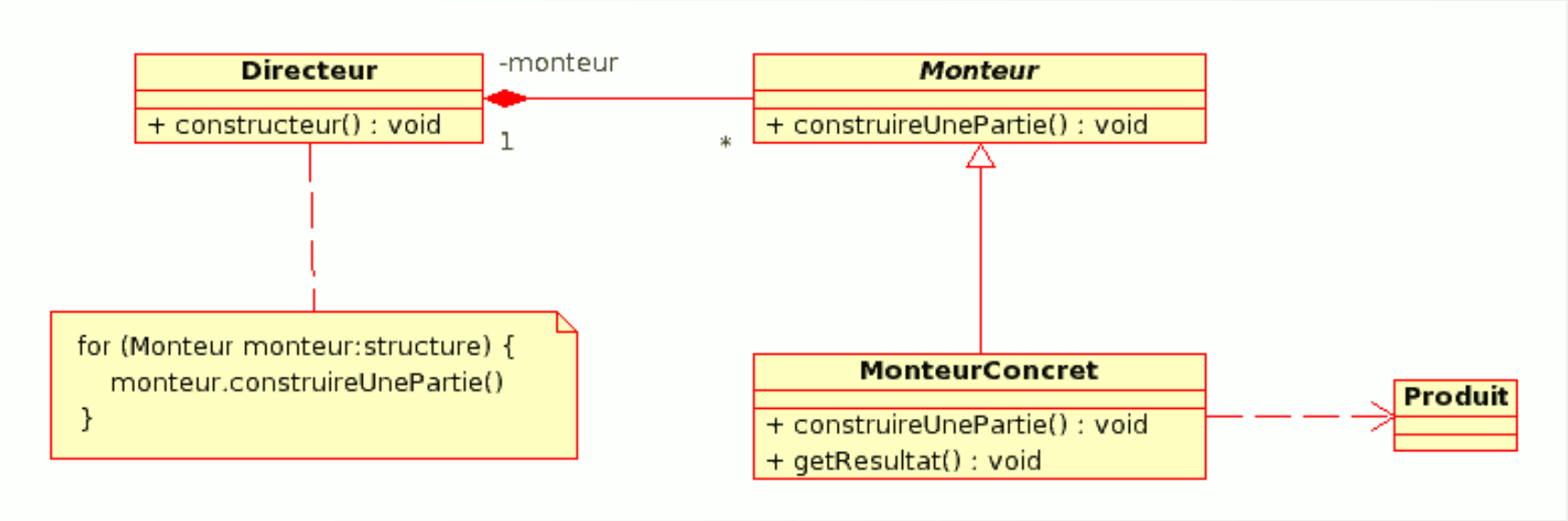
```
class Pizza {
  private String pate = "";
  private String sauce = "";
  private String garniture = "";

  public void setPate(String pate)          { this.pate = pate; }
  public void setSauce(String sauce)         { this.sauce = sauce; }
  public void setGarniture(String garniture) { this.garniture = garniture; }
}
```

# Monteur – modélisation UML

- Un exemple d'application



# Monteur – modélisation UML générique



# Patron Fabrique

- Problème/contexte
  - Créer un objet sans préciser explicitement la classe réelle qui sera instanciée
  - La classe exacte de l'objet n'est donc pas connue par l'appelant
- Solution
  - **définir une interface pour la création d'un objet, mais en laissant aux sous-classes le choix des classes à instancier**
  - **permettre à une classe de déléguer l'instanciation à des sous-classes**
- Implantations connues dans le *core* de Java
  - `java.util.Calendar#getInstance()`
  - `java.util.ResourceBundle#getBundle()`
  - `java.text.NumberFormat#getInstance()`
  - `java.nio.charset.Charset#forName()`

# Fabrique - Exemple de codage en Java

- Exemple simplifié

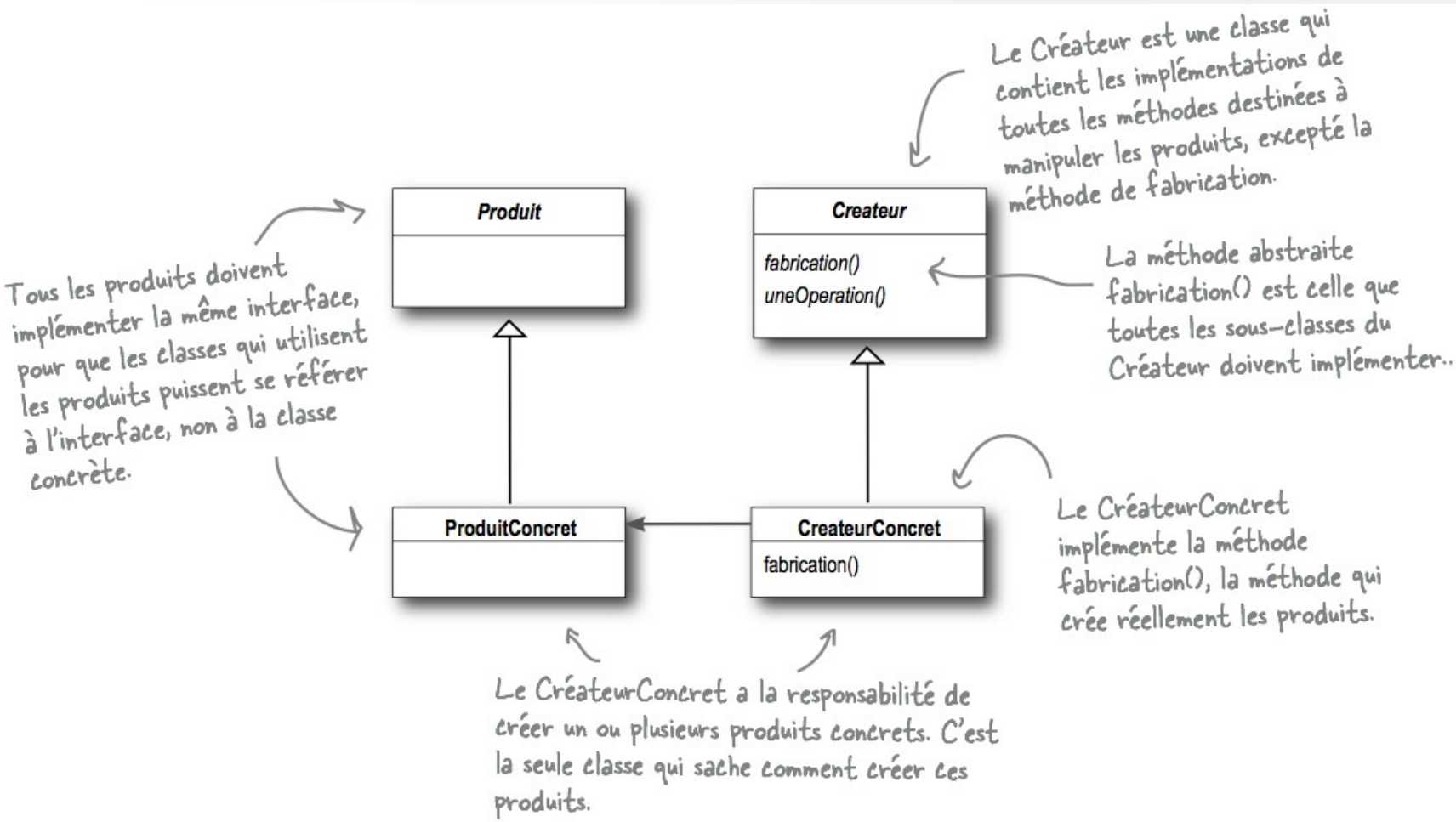
```
public class FabriqueAnimal {

  Animal getAnimal(String typeAnimal) throws ExceptionCreation {
    Animal animal;
    if("chat".equals(typeAnimal)) {
      animal = new Chat();
    } else if("chien".equals(typeAnimal)) {
      animal = new Chien();
    } else {
      throw new ExceptionCreation("Impossible de créer un " + typeAnimal);
    }
    return animal;
  }

}
```

# Fabrique - Exemple de codage en Java

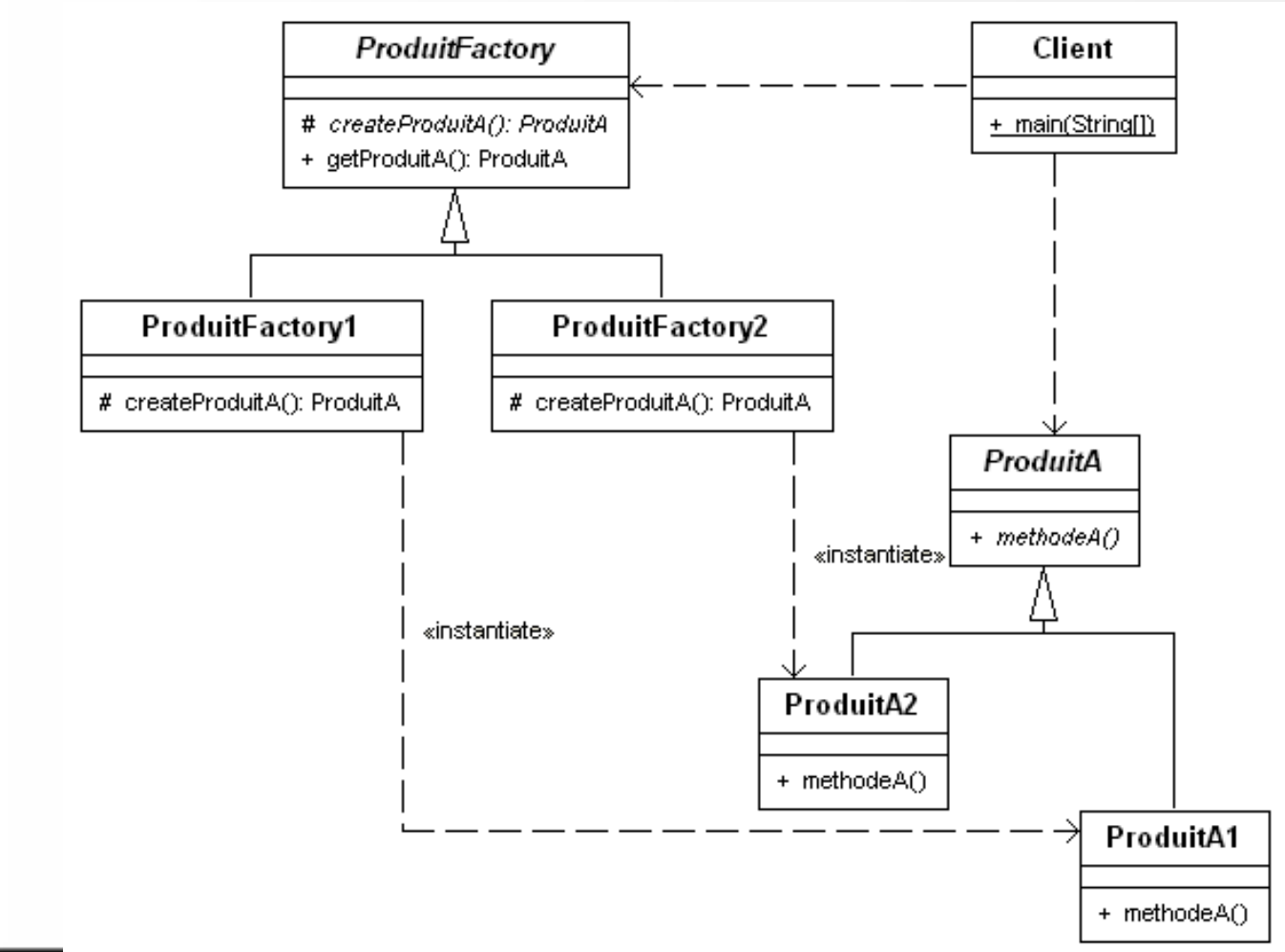
- Autre exemple simplifié (sans classe abstraite pour la fabrique)

```
public class MazeGame {
    public MazeGame() {
        Room room1 = makeRoom();
        Room room2 = makeRoom();
        room1.connect(room2);
        this.addRoom(room1);
        this.addRoom(room2);
    }

    protected Room makeRoom() {
        return new OrdinaryRoom();
    }
}

public class MagicMazeGame extends MazeGame {
    @Override
    protected Room makeRoom() {
        return new MagicRoom();
    }
}
```

# Fabrique – modélisation UML générique



# Fabrique – modélisation UML (avec plusieurs fabriques/produits concrets)



# Patron Fabrique abstraite

- Problème/contexte
  - Faciliter la création de différents objets groupés selon des critères communs
  - La classe exacte de l'objet n'est pas connue par l'appelant
- Solution
  - **Fournir une interface (par encapsulation d'un groupe de fabriques ayant une thématique commune) pour créer des familles d'objets apparentés ou dépendants sans avoir à spécifier leurs classes concrètes**
  - **Le code client crée une implémentation concrète de la fabrique abstraite, puis utilise les interfaces génériques pour créer des objets concrets de la thématique qu'il utilise à travers des interfaces génériques des objets produit**
- Implantations connues dans le *core* de Java
  - *Dans les librairies graphiques comme AWT/SWING…*
  - *Idem avec le look&feel*

# Fabrique abstraite - Exemple de codage en Java

- La fabrique abstraite et les fabriques concrètes

```
public abstract class GUIFactory {
    public static GUIFactory getFactory() {
         int sys = readFromConfigFile("OS_TYPE");

         if (sys == 0)
             return(new WinFactory());

         return(new OSXFactory());
    }
    public abstract Button createButton();
 }

 class WinFactory extends GUIFactory {
     public Button createButton(){
         return(new WinButton());
     }
 }

 class OSXFactory extends GUIFactory{
     public Button createButton(){
         return(new OSXButton());
     }
 }
```

# Fabrique abstraite - Exemple de codage en Java

- Le produit abstrait et les produits concrets

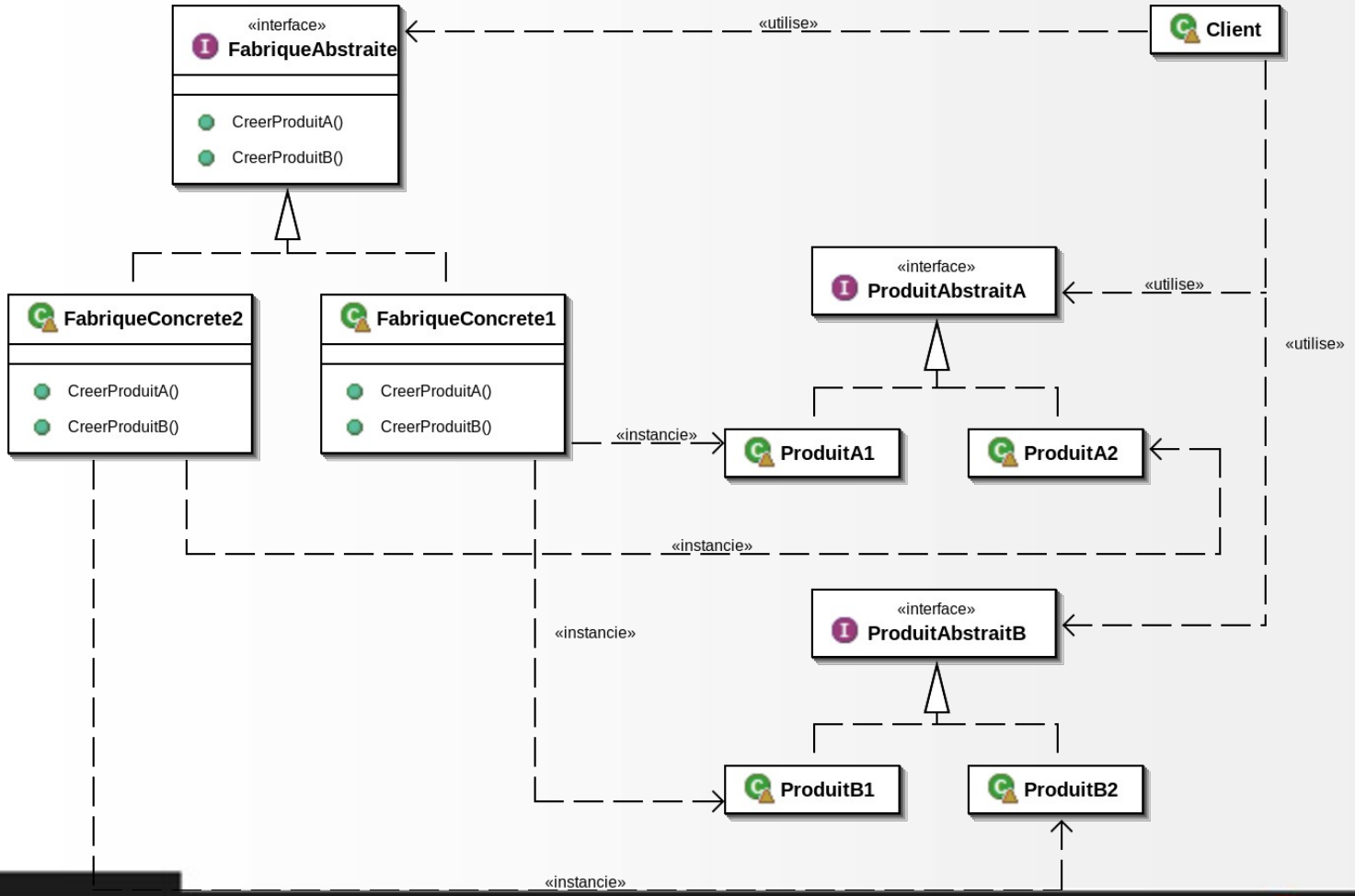
```
public abstract class Button {
     private String caption;

     public String getCaption() {
         return caption;
     }

     public void setCaption(String caption) {
         this.caption = caption;
     }

     public abstract void paint();
 }

 class WinButton extends Button{
     public void paint(){
         System.out.println("I'm a WinButton: "+ getCaption());
     }
 }

 class OSXButton extends Button{
     public void paint(){
         System.out.println("I'm an OSXButton: "+ getCaption());
     }
 }
```

# Fabrique abstraite - Exemple de codage en Java

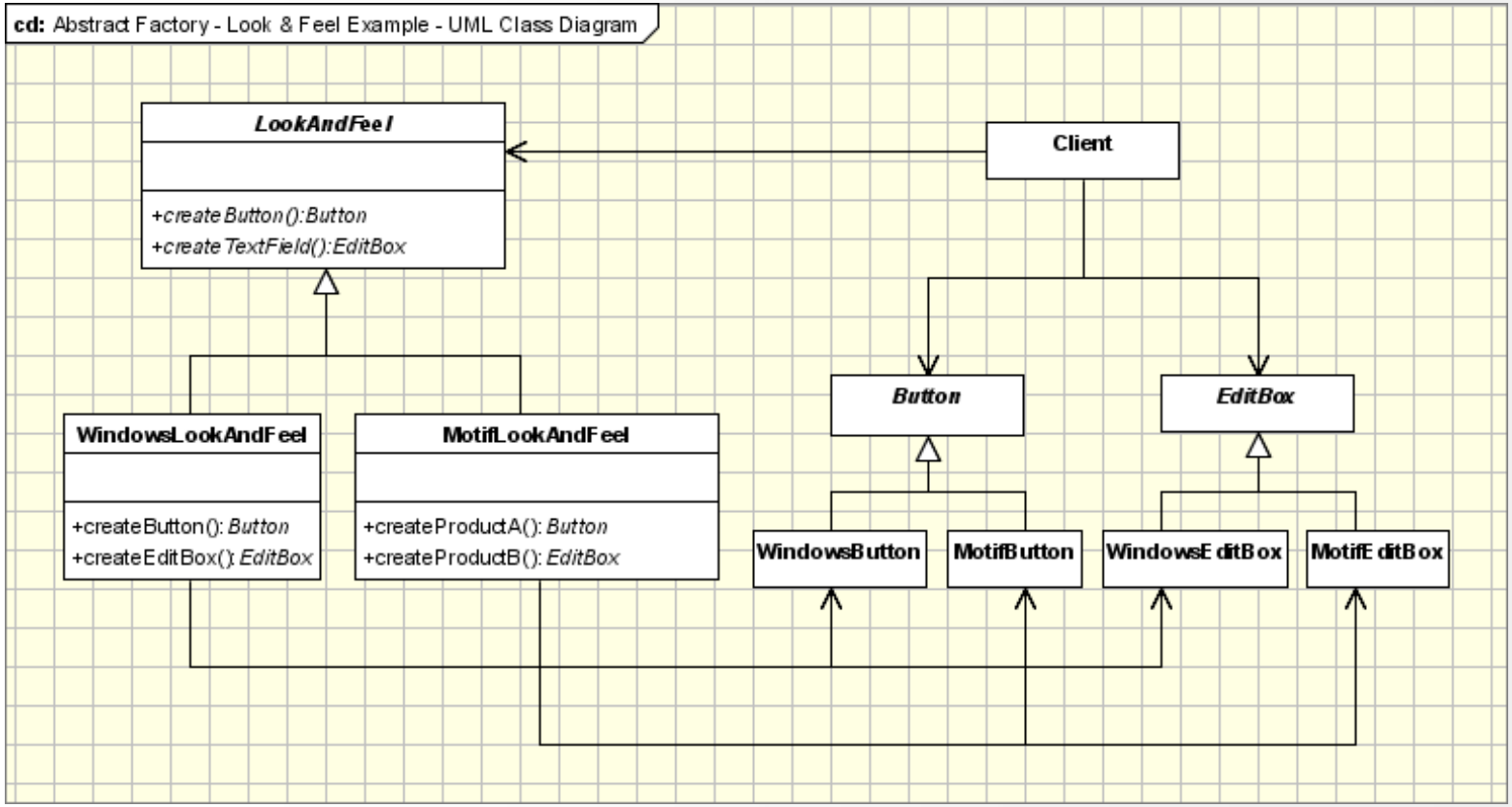
- Un exemple de code client

```
public class Application {
     public static void main(String[] args)  {
          GUIFactory aFactory = GUIFactory.getFactory();
          Button aButton = aFactory.createButton();
          aButton.setCaption("Play");
          aButton.paint();
     }

     //output is
     //I'm a WinButton: Play
     //or
     //I'm a OSXButton: Play
 }
```

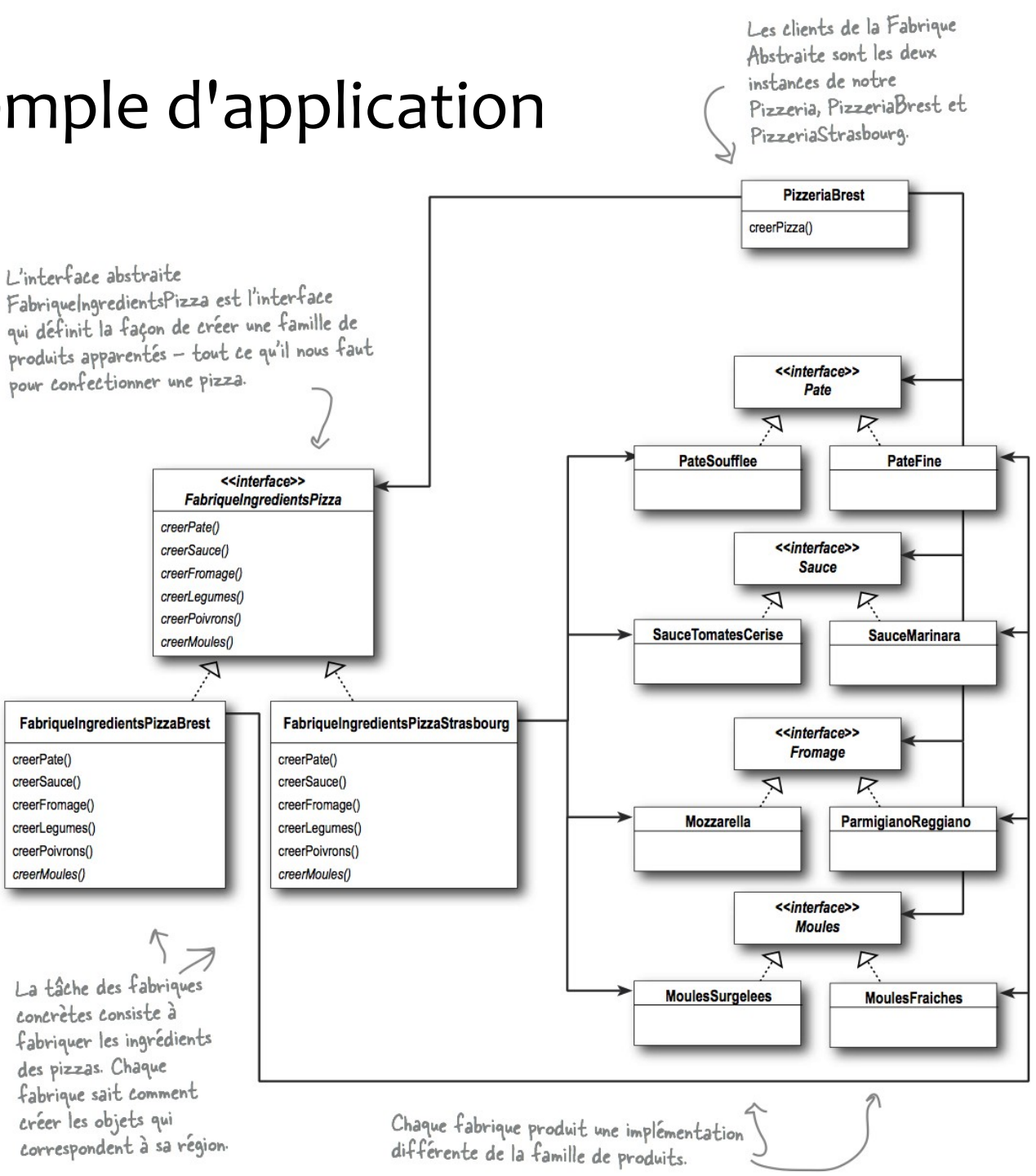
# Fabrique abstraite – modélisation UML générique



# Fabrique abstraite – exemple d'application



# Autre exemple d'application



# Fabrique abstraite

- Ce patron est donc une extension du patron **Fabrique**

- Voyez-vous le patron Fabrique dans la modélisation UML de Fabrique Abstraite ?

# Les patrons GoF structurels

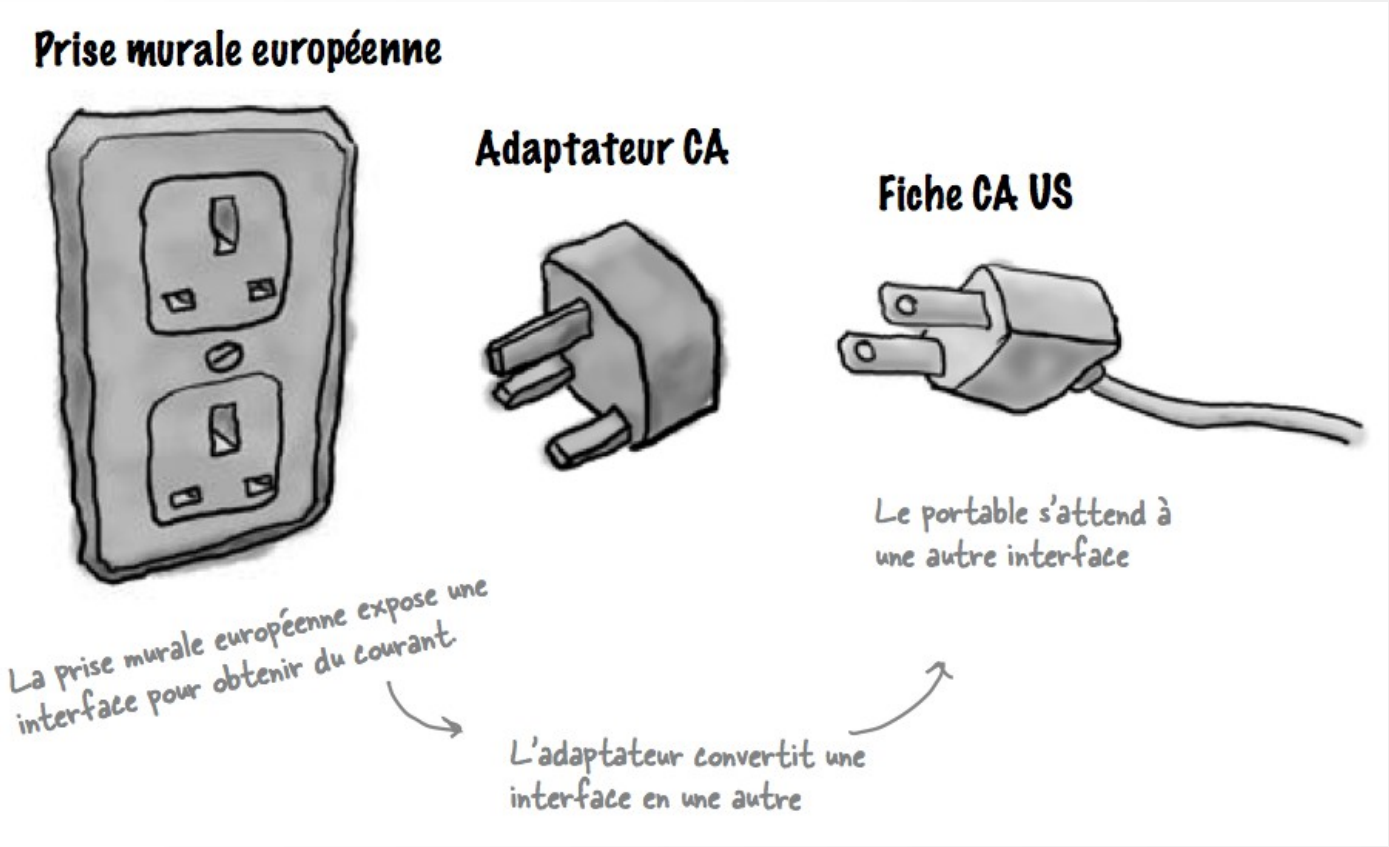
# L'adaptateur - introduction

- Mise en situation-problème

```
ComposantPasCompatible compo = new ComposantPasCompatible();

// compo.afficher()  n'existe pas...

IComposantAffichable compoAdapte = …….  ;

compoAdapte.afficher();
```

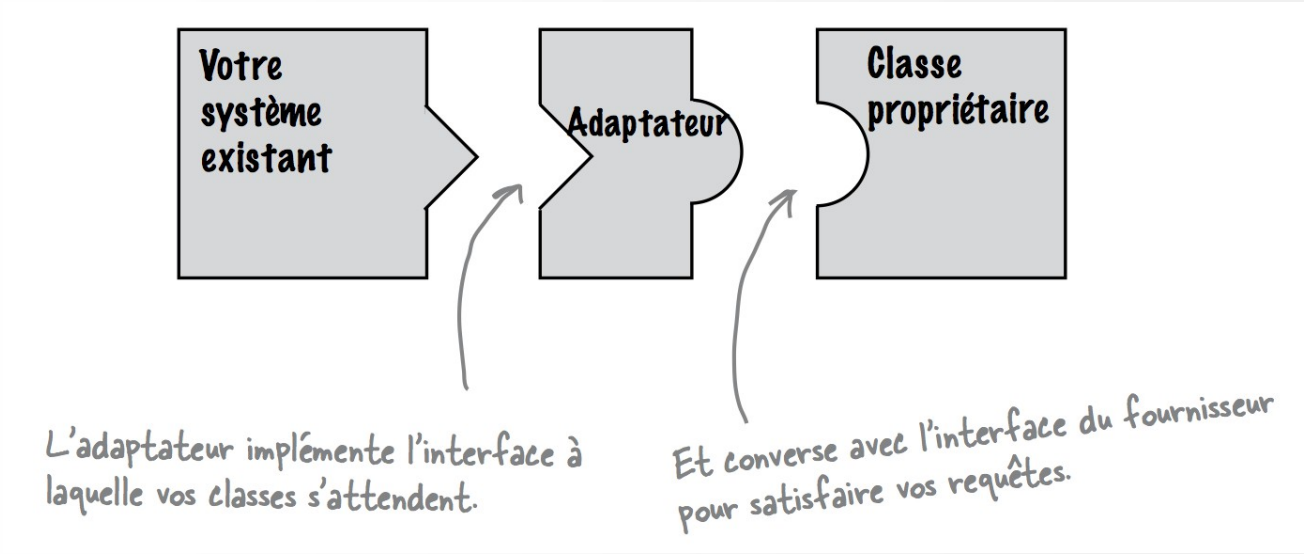
# L'adaptateur

- Un besoin du quotidien…



# L'adaptateur

- Schématisation d'une solution



# Patron Adaptateur

- Problème/contexte
  - Intégrer une classe que vous ne voulez/pouvez pas modifier
  - Faire collaborer des classes/objets dont les interfaces sont incompatibles
- Solution
  - **Convertir l'interface d'une classe en une autre conforme à celle du client**
- Implantations connues dans le *core* de Java
  - `java.util.Arrays#asList()`
  - `java.io.InputStreamReader(InputStream)` `(returns a Reader)`
  - `java.io.OutputStreamWriter(OutputStream)` `(returns a Writer)`

# Adaptateur - Exemple de codage en Java

- Version adaptateur d'objet

```
public class AdaptCompoPasCompatible implements IComposantAffichable {

    private ComposantPasCompatible comp;

    public AdaptCompoPasCompatible(ComposantPasCompatible compo) {
        comp = compo;
    }

    public void afficher() {
        comp.dessiner();
    }

}
```

Principe de délégation

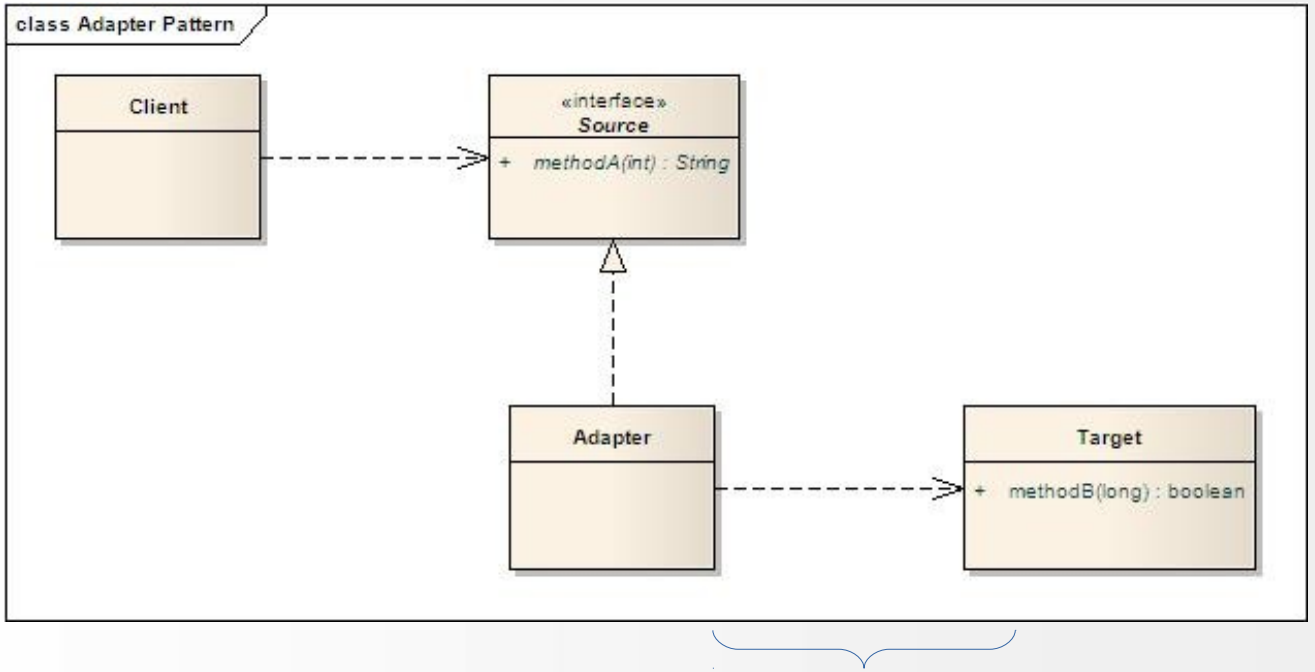
# Adaptateur - Exemple de codage en Java

- Version adaptateur de classe

```
public class AdaptClasseCompoPasCompatible
    extends ComposantPasCompatible implements IComposantAffichable {

    public void afficher() {
        dessiner();
    }

}
```

# Adaptateur

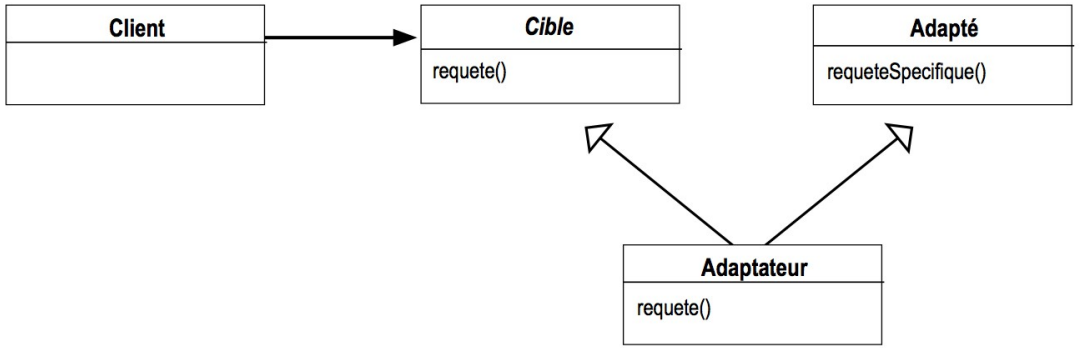
- Modélisation UML



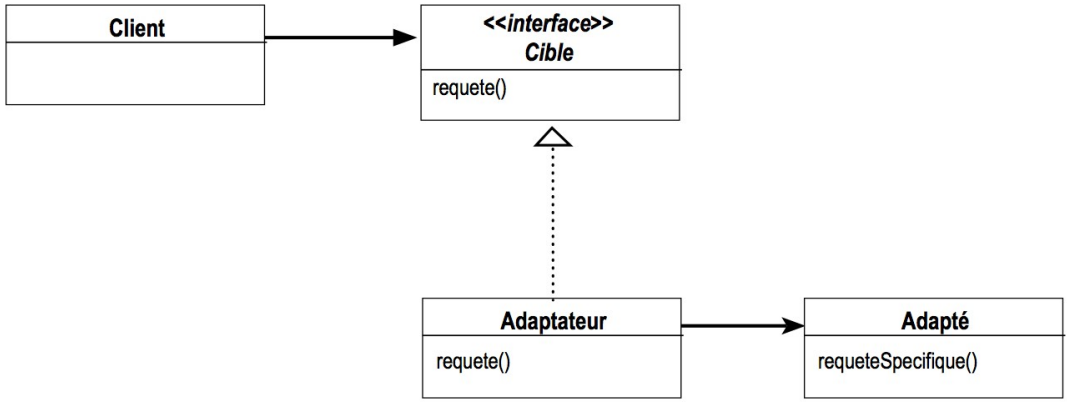

Par héritage (niveau classe) ou agrégation/délégation (niveau objet)

# L'adaptateur – 2 niveaux

**Adaptateur de classe**




**Adaptateur d'objet**



# Patron Composite

- Problème/contexte
  - manipuler un groupe d'objets de la même façon que s'il s'agissait d'un seul objet
  - Appliquer des traitements sur une structure de données « arbre » sans distinguer les feuilles des noeuds
- Solution
  - **utiliser une classe abstraite/interface factorisant les méthodes communes**
  - **Une sous-classe concrète a une collection vers l'élément abstrait (et les méthodes d'ajout/retrait correspondantes)**
- Implantations connues dans le *core* de Java
  - `java.awt.Container#add(Component)` `(practically all over Swing thus)`

# Composite - Exemples de codage en Java

- L'exercice de TD de M2103
  - Fichier / Dossier / Element

- Le dernier exercice de l'examen
  - Canard / Troupe / Cancaneur

# Composite - Exemple de codage en Java

```
interface Graphic {
    //Imprime le graphique.
    public void print();
}

class CompositeGraphic implements Graphic{
    //Collection de graphiques enfants.
    private List<Graphic> mChildGraphics = new ArrayList<Graphic>();

    //Imprime le graphique.
    public void print(){
        for (Graphic graphic : mChildGraphics){
            graphic.print();
        }
    }
     //Ajoute le graphique à la composition composition.
    public void add(Graphic graphic){
        mChildGraphics.add(graphic);
    }
     //Retire le graphique de la composition.
    public void remove(Graphic graphic){
        mChildGraphics.remove(graphic);
    }
}
```

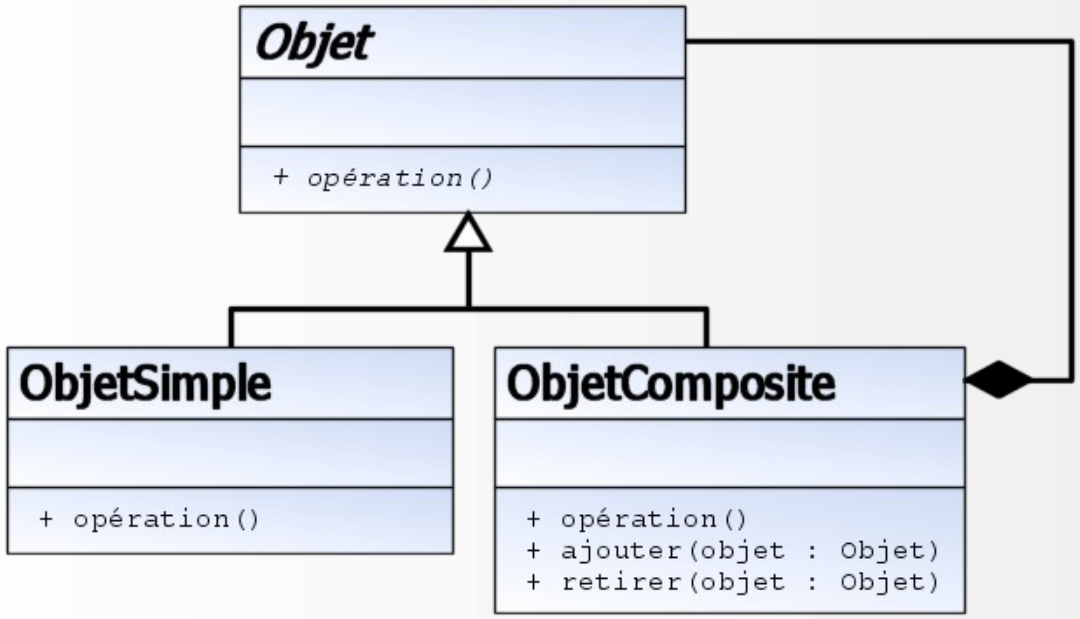
```
class Ellipse implements Graphic{
    //Imprime le graphique.
    public void print()    {
        System.out.println("Ellipse");
    }
}
```

# Composite - Exemple de codage en Java

```
...
        //Initialise quatre ellipses
        Ellipse ellipse1 = new Ellipse();
        Ellipse ellipse2 = new Ellipse();
        Ellipse ellipse3 = new Ellipse();
        Ellipse ellipse4 = new Ellipse();

        //Initialise three graphiques composites
        CompositeGraphic graphic = new CompositeGraphic();
        CompositeGraphic graphic1 = new CompositeGraphic();
        CompositeGraphic graphic2 = new CompositeGraphic();

        //Composes les graphiques
        graphic1.add(ellipse1);
        graphic1.add(ellipse2);
        graphic1.add(ellipse3);

        graphic2.add(ellipse4);

        graphic.add(graphic1);
        graphic.add(graphic2);

        //Imprime le graphique complet (quatre fois la chaîne "Ellipse").
        graphic.print();
...
```

# Composite

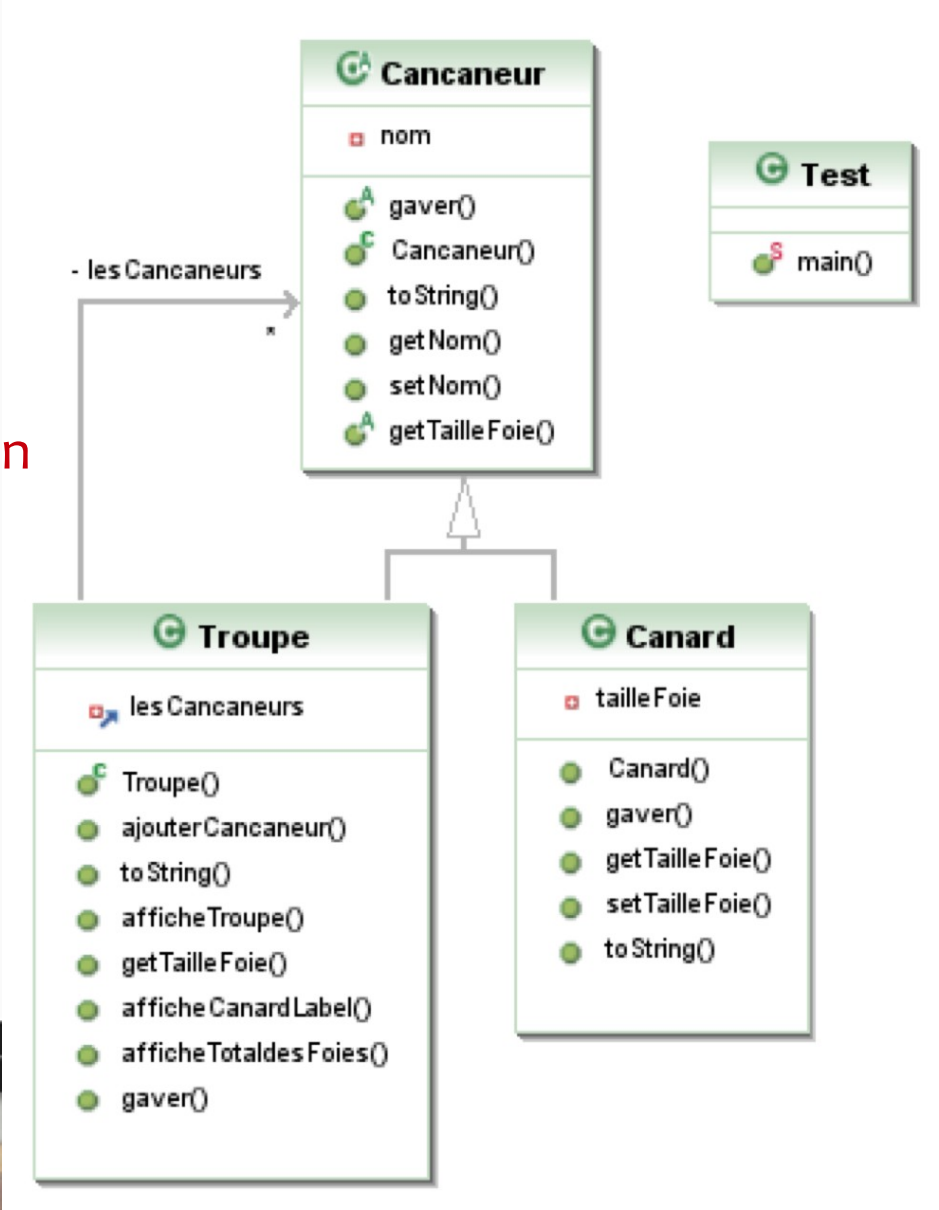
- Modélisation UML



# Composite

- Modélisation UML d'une application du patron

Quelle est l'opération commune ?



# Patron Décorateur

- Problème/contexte
  - Attacher dynamiquement de nouvelles responsabilités à un objet
  - Ajouter de nouveaux comportements/propriétés
  - Palier le manque de souplesse de l'héritage
- Solution
  - **Créer une adaptateur à niveau objet de l'objet à décorer**
  - **L'adaptateur étant composé avec l'interface de l'objet concret de manière à permettre de décorer des décorateurs**
- *Décorateur* ressemble à *Composite* mais diffère en terme de finalités
- Implantations connues dans le *core* de Java
  - *All subclasses of java.io.InputStream, OutputStream, Reader and Writer have a constructor taking an instance of same type.*

# Décorateur - Exemple de codage en Java

- L'objet concret et son interface

```
interface Voiture {
     public double getPrix();
}

class AstonMartin implements Voiture {

     public double getPrix() {
          return 999.99;
     }
}
```

# Décorateur - Exemple de codage en Java

- Le décorateur abstrait

```
abstract class Option implements Voiture {
     protected Voiture _originale;
     protected double _tarifOption;

     public Option(Voiture originale, double tarif) {
          _originale = originale;
          _tarifOption += tarif;
     }

     public double getPrix() {
          return _originale.getPrix() + _tarifOption;
     }
}
```

# Décorateur - Exemple de codage en Java

- Les décorateurs concrets

```
class VoitureAvecClimatisation extends Option {
     public VoitureAvecClimatisation(Voiture originale) {
          super(originale, 1.0);
     }
}

class VoitureAvecParachute extends Option {
     public VoitureAvecParachute(Voiture originale) {
          super(originale, 10.0);
     }
}

class VoitureAmphibie extends Option {
     public VoitureAmphibie(Voiture originale) {
          super(originale, 100.0);
     }
}
```

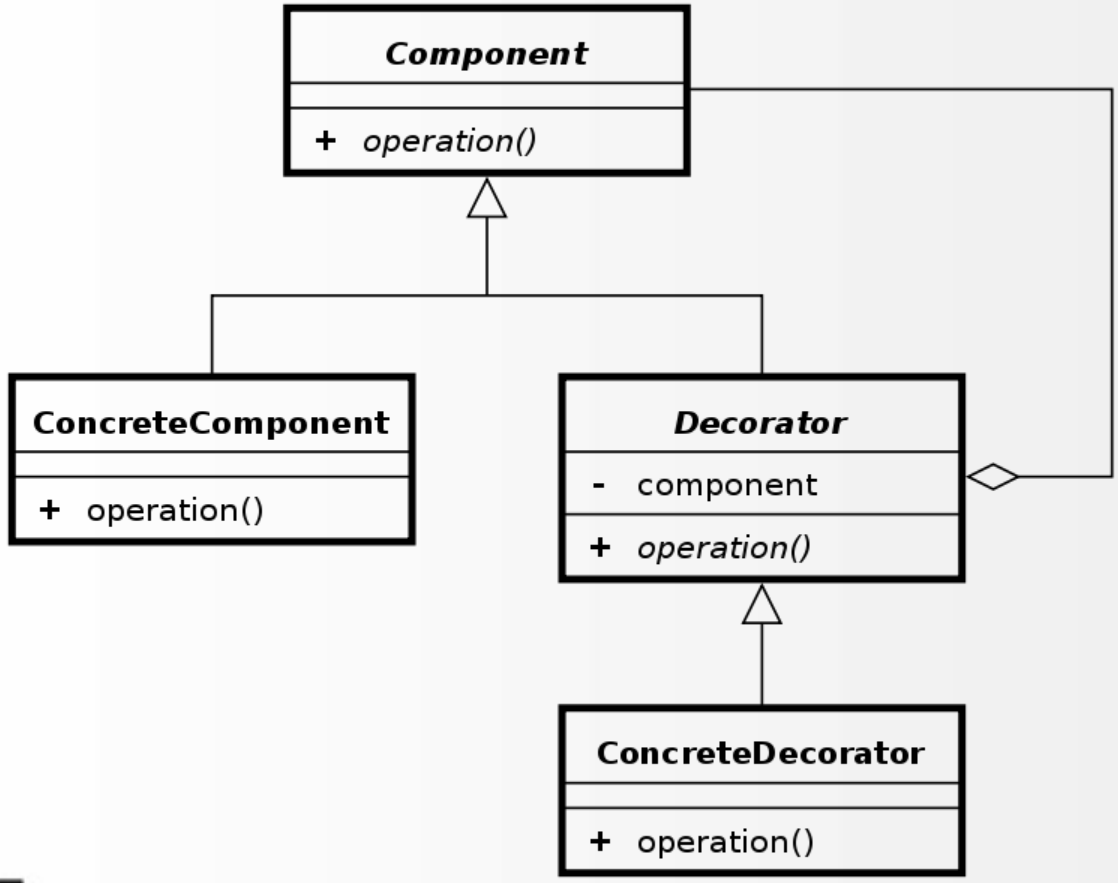
# Décorateur - Exemple de codage en Java

- La classe cliente

```
public class Usine {

  public static void main(String[] args) {

     // voiture de serie avec clim
    Voiture astonMartin =  new VoitureAvecClimatisation(new AstonMartin());

     // Ajout d'une option (décorateur)
    astonMartin = new VoitureAvecParachute(astonMartin);
    astonMartin = new VoitureAmphibie(astonMartin);

    System.out.println(String.format("Prix = %.2f€", astonMartin.getPrix()));
    // affiche "Prix = 1110,99€"
  }
}
```

# Décorateur

- Modélisation UML



# Patron Pont

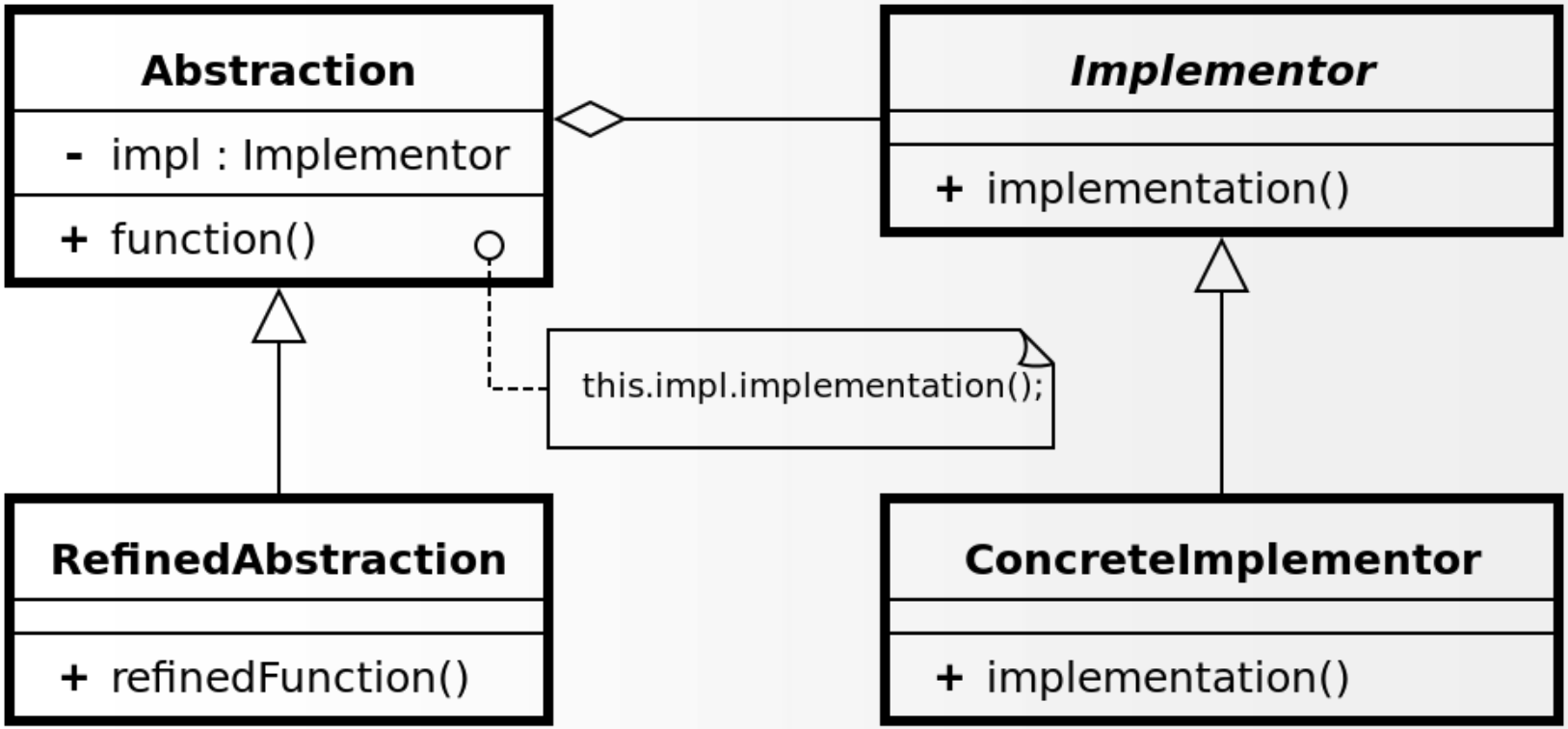
- Problème/contexte
  - Découpler l'interface d'une classe et son implémentation afin que abstraction et implémentation soient indépendants et puissent varier
- Solution
  - **Utiliser l'encapsulation, l'agrégation, et l'héritage pour séparer les responsabilités des différentes classes**

- *Pont* est assez proche de *l'adaptateur* : ne pas confondre !

- Pas d'implantations connues dans le *core* de Java

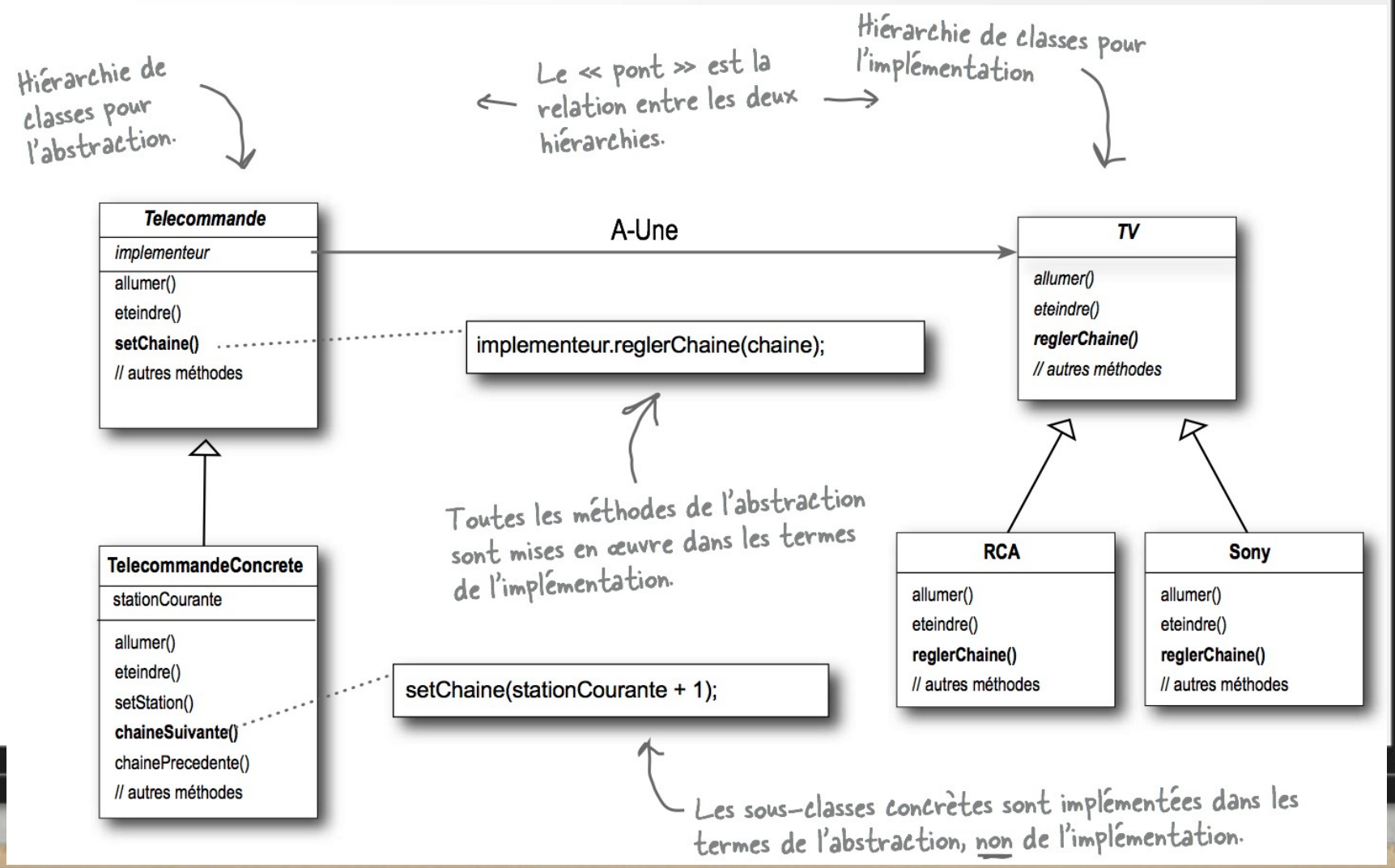
# Pont

- Modélisation UML



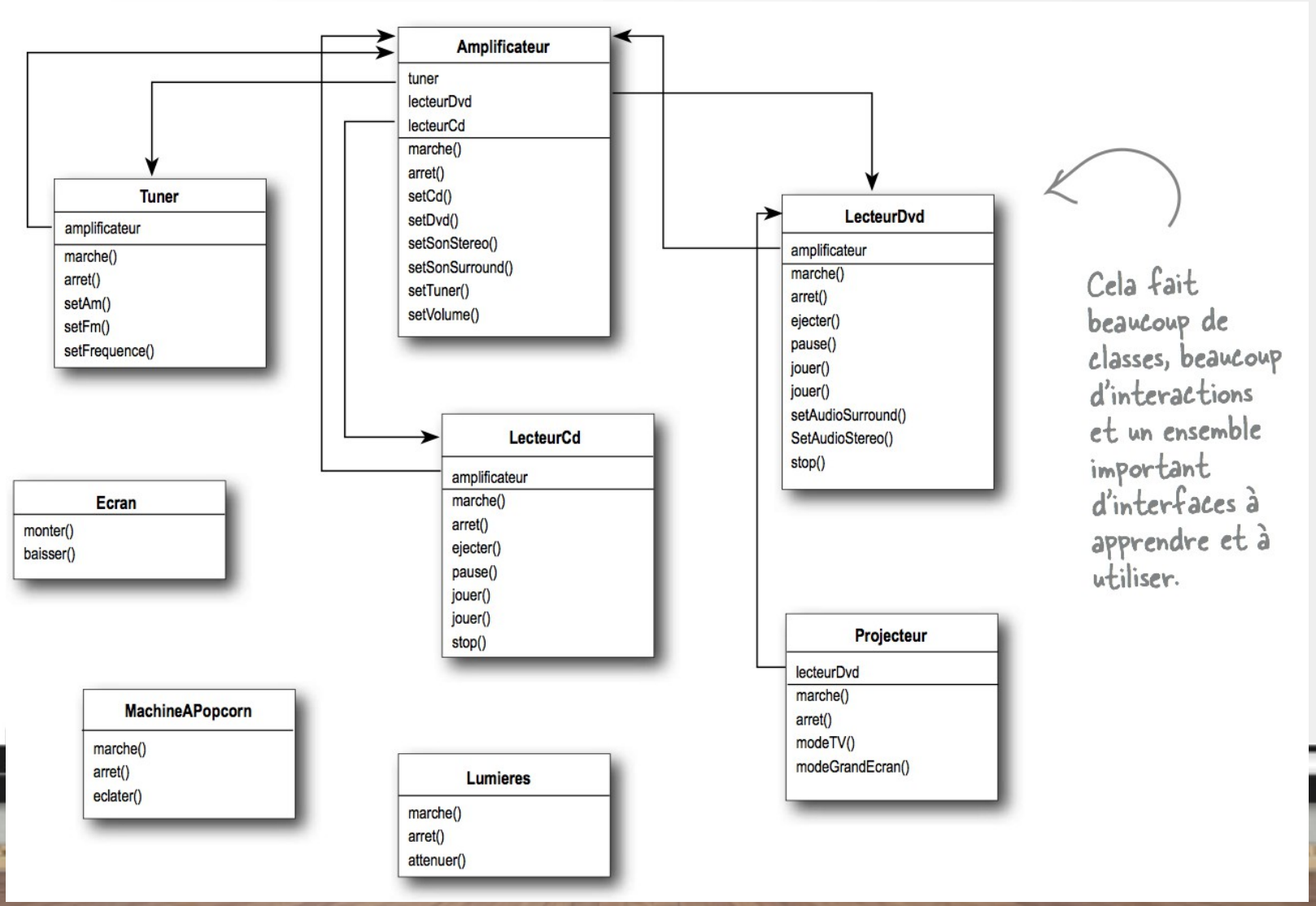
# Pont

- Modélisation UML d'une application



# Façade

- Problème introductif : un Home Cinéma de luxe…



# Façade

- Du point de vue du code...

```
machineAPopCorn.marche();
machineAPopCorn.eclater();
lumieres.attenuer(10);
ecran.baisser();
projecteur.marche();
projecteur.setEntree(dvd);
projecteur.modeGrandEcran();
amp.marche();
amp.setDvd(dvd);
amp.setSonSurround();
amp.setVolume(5);
dvd.marche();
dvd.jouer(film);
```

Beaucoup d'objets avec lesquels interagir pour regarder un film !

# Patron Façade

- Problème/contexte
  - Faciliter l'utilisation d'un système complexe comprenant de nombreux objets en interactions
  - Découpler un client d'un sous-système de composants
- Solution
  - **Fournir une interface unifiée à l'ensemble des interfaces d'un sous-système**
  - **Fournir une interface de plus haut niveau qui rend le sous-système plus facile à utiliser**
- *Façade* est assez proche de *l'adaptateur* : mais le premier vise à simplifier une interface tandis que le second est de la convertir en quelque chose de différent
- Implantations connues dans le *core* de Java
  - `javax.faces.context.FacesContext`

# Façade – retour à l'exemple de codage en Java

- La classe façade

```
public class FacadeHomeCinema {
     Amplificateur amp;
     Tuner tuner;
     LecteurDvd dvd;
     LecteurCd cd;
     Projecteur projecteur;
     Lumieres lumieres;
     Ecran ecran;
     MachineAPopcorn machineAPopCorn;

     public FacadeHomeCinema(Amplificateur amp, Tuner tuner, LecteurDvd dvd, LecteurCd cd,
          Projecteur projecteur, Ecran ecran,Lumieres lumieres, MachineAPopcorn
             machineAPopCorn) {
          this.amp = amp;
          this.tuner = tuner;
          this.dvd = dvd;
          this.cd = cd;
          this.projecteur = projecteur; this.ecran = ecran;
          this.lumieres = lumieres; this.machineAPopCorn = machineAPopCorn;
     }
// autres méthodes
}
```

# Façade – retour à l'exemple de codage en Java

- Quelques méthodes de la façade

```
public void regarderFilm(String film) {
    System.out.println("Vous allez voir un bon film...");
    machineAPopCorn.marche();  machineAPopCorn.eclater();
    lumieres.attenuer(10) ;
    ecran.baisser();
    projecteur.marche();  projecteur.modeGrandEcran();
    amp.marche();
    amp.setDvd(dvd);
    amp.setSonSurround();
    amp.setVolume(5);
    dvd.marche() ;
    dvd.jouer(film);
}
public void arreterFilm() {
    System.out.println("C'est la fin du film...");
    machineAPopCorn.arret();
    lumieres.marche();
    ecran.monter();
    projecteur.arret();
    amp.arret();
    dvd.stop(); dvd.ejecter();  dvd.arret();
}
```

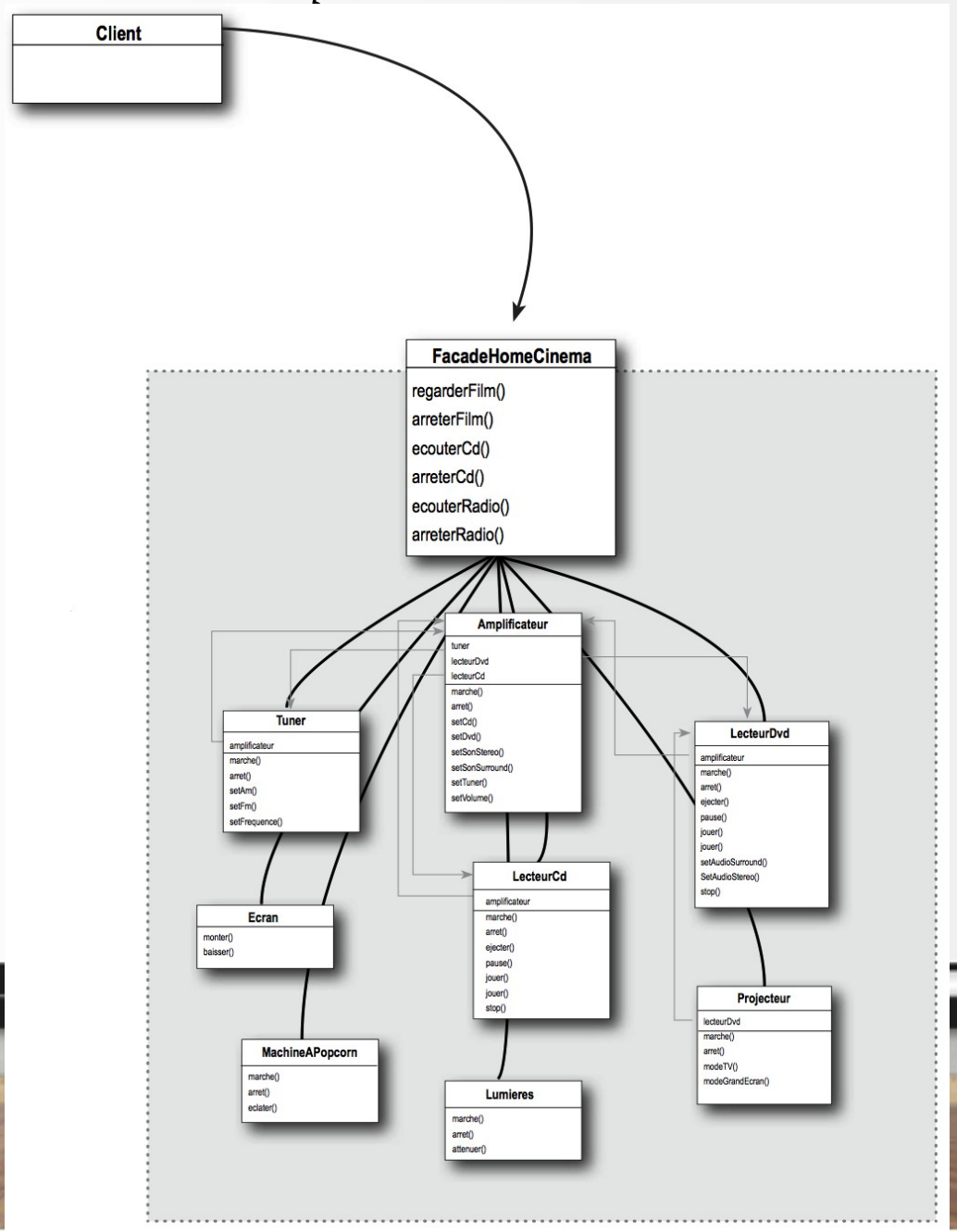
# Façade – retour à l'exemple de codage en Java

- Le code client

```
public class TestHomeCinema {
    public static void main(String[] args) {
        // instanciation des composants
        FacadeHomeCinema homeCinema = new FacadeHomeCinema(amp, tuner, dvd, cd,
            projecteur, ecran, lumieres, machineAPopCorn);

        homeCinema.regarderFilm("Hôtel du Nord");
        homeCinema.arreterFilm();
    }
}
```

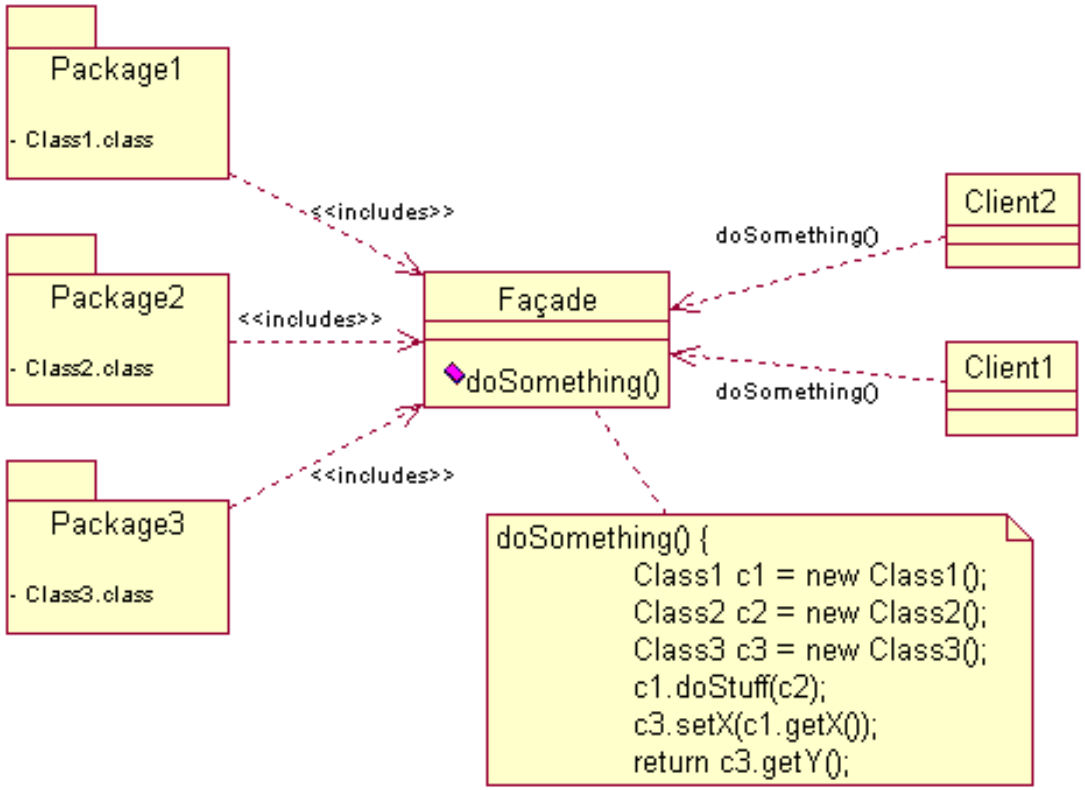
# Façade

- Modélisation UML de l'exemple



# Façade

- Modélisation UML générale



# Patron Proxy

- Problème/contexte
  - Contrôler l'accès à un autre objet qui peut être distant, coûteux à créer ou qui doit être sécurisé
- Solution
  - **Fournir un remplaçant à un autre objet, pour en contrôler l'accès**
- Implantations connues dans le *core* de Java
  - *`java.lang.reflect.Proxy`*
  - *`java.rmi.*, the whole API actually`*

# Proxy – exemple de codage en Java

```
interface Image {
    public void displayImage();
}

//on System A
class RealImage implements Image {

    private String filename = null;

    public RealImage(final String filename) {
        this.filename = filename;
        loadImageFromDisk();
    }

    private void loadImageFromDisk() {
        System.out.println("Loading   " + filename);
    }

    public void displayImage() {
        System.out.println("Displaying " + filename);
    }

}
```

L'objet coûteux

# Proxy – exemple de codage en Java

Le proxy

```
//on System B
class ProxyImage implements Image {

    private RealImage image = null;
    private String filename = null;

    public ProxyImage(final String filename) {
        this.filename = filename;
    }

    public void displayImage() {
        if (image == null) {
           image = new RealImage(filename);
        }
        image.displayImage();
    }

}
```

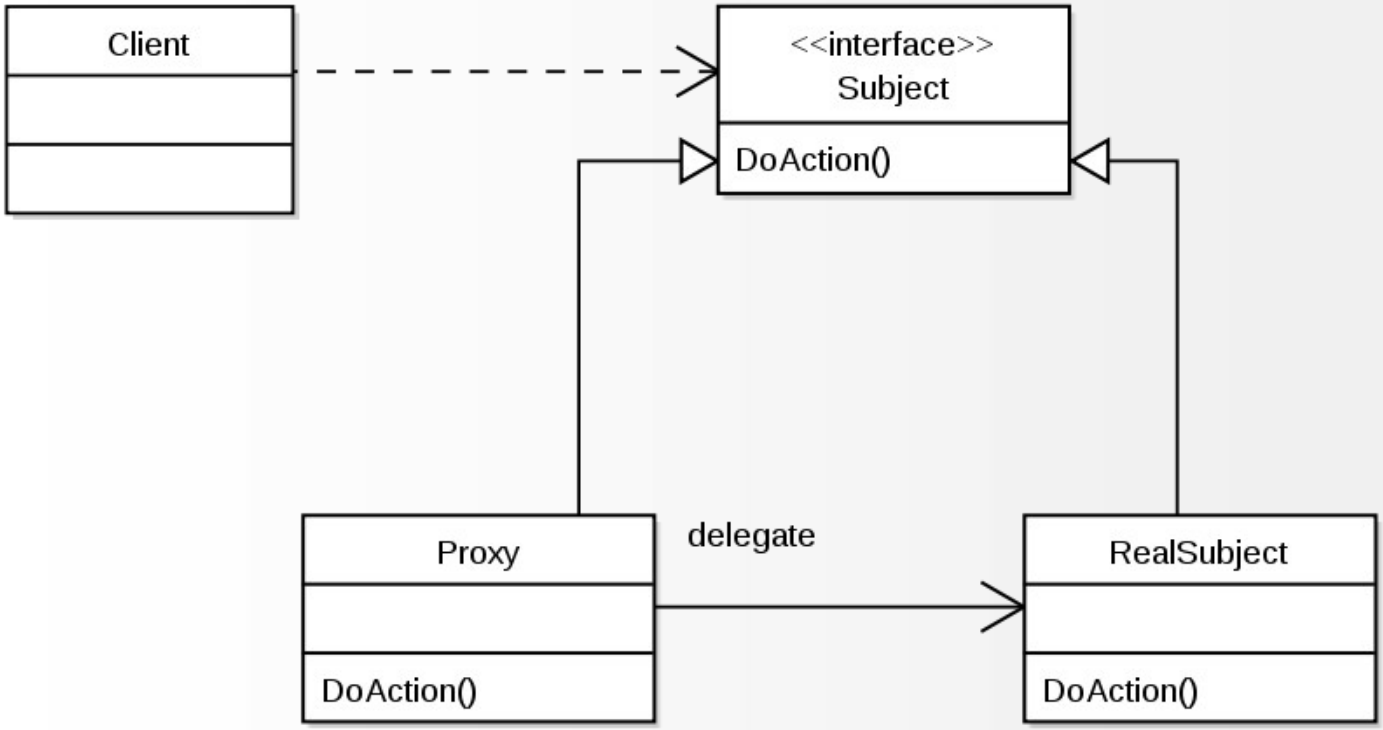
# Proxy – exemple de codage en Java

Le client

```
class ProxyExample {


   public static void main(String[] args) {
        final Image IMAGE1 = new ProxyImage("HiRes_10MB_Photo1");
        final Image IMAGE2 = new ProxyImage("HiRes_10MB_Photo2");

        IMAGE1.displayImage(); // loading necessary
        IMAGE1.displayImage(); // loading unnecessary
        IMAGE2.displayImage(); // loading necessary
        IMAGE2.displayImage(); // loading unnecessary
        IMAGE1.displayImage(); // loading unnecessary
    }

}
```

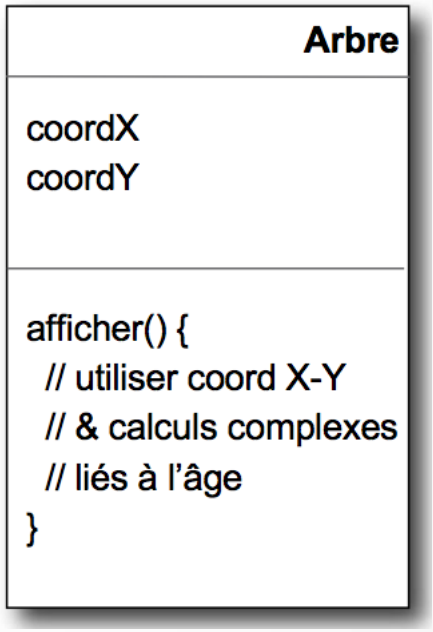
# Proxy

- Modélisation UML générale



# Poids-mouche

- Problème introductif : l'application paysagère
  - Un objet arbre gère ses propres coordonnées et son affichage
  - L'utilisation d'un nombre très important d'arbres finit par ralentir l'application...



# Patron Poids-mouche

- Problème/contexte
  - Réduire le nombres d'instances de petite taille présentes en mémoire pour économiser de l'espace mémoire
- Solution
  - **N'utiliser qu'une seule instance de l'objet de petite taille**
  - **Utiliser un objet « gestionnaire » qui maintient l'état de TOUS les objets « virtuels »**
- Implantations connues dans le *core* de Java
  - *`java.lang.Integer#valueOf(int)`*

# Poids-mouche

- Modélisation UML de l'exemple