

DUT-Info/S3/M3105 (CPOA)

Jean-Michel Bruel

Version 1.4, 2016-11-16

1. Introduction

Ce cours porte sur la Conception et Programmation Objet Avancée.

1.1. Concepts objets

Vous avez appris (cf. M2103 et M2104) un certain nombre de **concepts objets** :

- Abstraction
- Encapsulation
- Héritage
- Polymorphisme

1.1.1. Abstraction

Définition (restrictive) :

Une classe est une **abstraction** des caractéristiques communes d'un ensemble d'objets.

1.1.2. Encapsulation

Définition (restrictive) :

Dans la description d'un objet, le but de l'**encapsulation** est de masquer les attributs et les méthodes, c'est à dire, la manière dont est réalisé le comportement de l'objet.

1.1.3. Héritage

Définition (simpliste) :

L'héritage est la transmission de caractéristiques à ses descendants.

La classe qui hérite dispose des méthodes et attributs de niveau **public** et **protected** de sa classe mère.

1.1.4. Polymorphisme

Le nom de polymorphisme vient du grec :

qui peut prendre plusieurs formes



L'héritage concerne les classes, le polymorphisme concerne les objets.

On distingue généralement trois types de polymorphisme :

- Le polymorphisme **ad hoc** (également surcharge ou en anglais overloading)
- Le polymorphisme **paramétrique** (également généricté ou en anglais template)
- Le polymorphisme **d'héritage** (également redéfinition, spécialisation ou en anglais overriding)

Polymorphisme ad hoc

- Appelé aussi **surcharge**.
- Permet d'avoir des fonctions de même nom dans des classes sans aucun rapport entre elles.
- Permet de définir des opérateurs d'utilisation différente en fonction des paramètres.

Le polymorphisme paramétrique

Appelé aussi **généricité**.

```
int method(int, int);
int method(int);
int method(float, float);
```

Le polymorphisme d'héritage

- Appelé aussi **spécialisation** (ou redéfinition).
- Lié à la redéfinition des méthodes héritées.

1.2. Objectifs de la conception objet

On essaye d'éviter trois problèmes principaux du développement :

- la rigidité
- la fragilité
- l'immobilité

1.2.1. La rigidité

Anticiper les évolutions susceptibles d'impacter l'application.

1.2.2. La fragilité

Eviter les erreurs provoquées par la modification d'une partie du code.

1.2.3. L'immobilité

Rendre moins difficile l'extraction d'une partie du code.

1.3. Bonnes pratiques et patrons

Pour répondre aux problèmes ci-dessus, on va s'attaquer à diminuer les **dépendances** et éviter l'"effet spaghetti".

Les qualités recherchées sont :

- Robustesse : les changements n'introduisent pas de régressions.
- Extensibilité : il est facile d'ajouter de nouvelles fonctionnalités.
- Réutilisabilité : il est possible de réutiliser certaines parties de code pour construire d'autres applications.

Nous allons apprendre des **bonnes pratiques** :

- Identifier les aspects qui varient et les séparer des aspects constants
- Programmer une interface, non une implémentation
- Préférer la composition à l'héritage
- Les classes doivent être ouvertes à l'extension, mais fermées à la modification
- Dépendez d'abstractions. Ne dépendez pas de classes concrètes (inversion des dépendances)
- Ne parlez pas aux inconnus

L'étape suivante consiste à apprendre les bonnes solutions de conception, ce qu'on appelle les **patrons de conception** (ou *design patterns* en anglais).

1.4. Organisation du cours



Rappel du rythme : 1 cours, 1 TD et 2 TPs par semaine. Pendant 8 semaines.

- La première semaine est consacrée au principe des patrons de conception, en partant d'un exemple (cours en fin de semaine).
- Les 5 ou 6 suivantes sont consacrées à l'étude de certains patrons classiques. Mise en pratique sur des exercices en TP.



Les TPs sont décalés d'une semaine (conception et étude d'un ou plusieurs patrons semaine N et mise en oeuvre en TP semaine N+1).

- Les 2 ou 3 suivantes, les étudiants en mode projet pour faire du *refactoring* d'applications réelles (conception aidée en TD sur les modèles UML™, mise en oeuvre en TP).

Voici une proposition de déroulement des semaines :

Semaine 1

SuperCanard, le grand classique, [Stratégie]

Semaine 2

- [Singleton]

Semaine 3

Patrons [Fabrique], [Proxy], [Etat]

Semaine 4

- [Observateur]
 - version intuitive (2 interfaces)
 - version Java (classe **Observable**)

Semaine 5

L'exemple de Meyer : menus en objet

Semaine 6

- Patrons Décorateur, Façade, Visiteur
- MVC avec l'exemple **JTable** de Java
- Patrons Chaîne de responsabilité (juste en cours)

Semaines 7 et 8

- Quelques idées de projet final :
 - Refactorer un code généré par **Umpire**.
 - Refactorer le code de MPA (mais pas le leur, celui d'un autre groupe)

1.5. Evaluation et notation

Comme prévu par le [planning des contrôles](#), les étudiants auront :

- une note de projet (TPs + projet final) ⇒ coef. 1
- une note d'examen final (semaine 5) ⇒ coef. 2

2. Rappels sur des éléments Java importants

2.1. Importance du typage

2.1.1. Différents types de typage

Le fait d'attribuer un type (une classe) à une variable (un objet) peut se faire de plusieurs façons :

- statique
- dynamique
- *duck typing*

2.1.2. Typage statique

On parle de **typage statique** quand la majorité des vérifications de type sont effectuées au moment de la **compilation**.

Exemple de typage statique

```
int i = 0; // cette déclaration indique explicitement que  
           // la variable i est de type entier
```

2.1.3. Typage dynamique

Le **typage dynamique** consiste à laisser l'ordinateur réaliser l'opération de typage *à la volée*, lors de l'**exécution du code**.

Exemple de typage dynamique

```
/**  
 * @author André Peninou  
 */  
public class Type {  
    void m() {  
        System.out.println ("Type");  
    }  
}  
public class SousType extends Type {  
    void m() {  
        System.out.println ("SousType");  
    }  
    void autreM(){  
        System.out.println ("Spécifique SousType");  
    }  
}  
...  
Type a = new Type();  
a.m(); // "Type"  
  
a = new SousType();  
a.m(); // "SousType"  
// Statique : a est un Type (à la compil)  
// Dynamique : a est un SousType au runtime.  
  
// D'où :  
a = new SousType();  
a.autreM();  
// NOK car type statique == A => autreM() n'existe pas à la compilation  
...
```

2.1.4. Duck typing

Style de **typage dynamique** où la **sémantique** d'un objet (c'est-à-dire son type) est déterminée par l'ensemble de ses **méthodes** et de ses **attributs**, et non par un type défini et nommé explicitement par le programmeur.

L'origine de cette expression est liée à cette citation :



Si je vois un animal qui vole comme un canard, cancane comme un canard, et nage comme un canard, alors j'appelle cet oiseau un canard.

— James Whitcomb Riley

Exemple de duck typing en Ruby

```
def calcule(a, b, c)
  return a*b+c
end

$a = calcule(6, 3, 2)
$b = calcule('6', 3, ', the number of the beast')

puts $a.to_s
puts $b.to_s
```

Ce qui donne :

```
20
666, the number of the beast
```



Pour aller plus loin : http://fr.wikipedia.org/wiki/Duck_typing

2.2. Importance de la visibilité

Dès que l'on commence à avoir une application conséquente, l'organisation en *package* devient obligatoire. Revenons donc sur les questions de **visibilité** des propriétés et méthodes, qui seront importants dans la plupart des aspects de ce module.

Si un champ d'une classe A :

- est *private*, il est accessible uniquement depuis sa propre classe ;
- à la visibilité *package* (visibilité par défaut, pas de mot-clef), il est accessible de partout dans le paquetage de A mais de nulle part ailleurs ;
- est *protected*, il est accessible de partout dans le paquetage de A et, si A est publique, grossièrement dans les classes héritant de A dans d'autres paquetages ;

- est *public*, il est accessible de partout dans le paquetage de A et, si A est publique, de partout ailleurs.



Ci-dessus, les niveaux de visibilité sont rangés par visibilité croissante.

```
package UN;
public class A {
    protected String attrprotected;
    String attrfriend; // friend
}
```

Si on définit une deuxième classe dans le même package :

```
package UN;
class B {
    ...
{
    A a = new A ();
    a.attrprotected// OK : même si bizarre
    a.attrfriend // OK : visible package
}
}

package UN;
class C extends A {
    ...
{
    this.attrprotected// OK : normal
    this.attrfriend // OK : visible package
}
}
```

```

package DEUX;
class B {
    ...
    {
        A a = new A ();
        a.attrprotected// NON OK : normal
        a.attrfriend // NON OK : normal, proche de "private"
    }
}

class C extends A {
    ...
    {
        this.attrprotected// OK : normal car protected et héritage
        this.attrfriend // NON OK : normal, proche de "private"
    }
}

```

À la question **private** ou **protected** ? Quel est le mieux pour les attributs ?

- C'est une question de **style de programmation !**
- Puristes (Meyer) ⇒ **private**
- Parfois utile : cf. *Strategy*, évite les getters/setters



Il n'y a pas de visibilité par défaut en **UML™**.

2.3. Retour sur les Membres **static**

```

class VariableDemo
{
    static int count=0;
    public void increment()
    {
        count++;
    }
    public static void main(String args[])
    {
        VariableDemo obj1=new VariableDemo();
        VariableDemo obj2=new VariableDemo();
        obj1.increment();
        obj2.increment();
        System.out.println("Obj1: count is="+obj1.count);
        System.out.println("Obj2: count is="+obj2.count);
    }
}

```

Output:

```
Obj1: count is=2
Obj2: count is=2
```

2.3.1. Membres **static** (suite)

Comment ça marche :

- Les variables **static** sont initialisées au chargement de la classe.
- Les variables **static** d'une classe sont initialisées avant que la moindre instance ne soit créée.
- Les variables **static** sont initialisées avant que la moindre méthode **static** ne s'exécute.

2.3.2. Méthodes **static**

```
import java.lang.Math;

class Another {
    public static void main(String[] args) {
        int result;

        result = Math.min(10, 20); //calling static method min by writing class name

        System.out.println(result);
        System.out.println(Math.max(100, 200));
    }
}
```

2.3.3. Méthodes **static** et appel aux méthodes non-statiques

```
public class Main {
    public static void main(String[] args) {
        Main p = new Main();
        k();
    }

    protected Main() {
        System.out.print("1234");
    }

    protected void k() {
    }
}
```

À l'exécution :

```
Main p = new Main(); // => prints 1234  
k()           // => raises error
```

Static method cannot call non-static methods

Bien sûr que si, sauf qu'il faut que cette dernière **porte sur une instance** de la classe.

Constructors are kind of a method with no return type.

En fait il vaudrait mieux les considérer comme une sorte de méthode statique. En effet elle ne requièrent pas de porter sur un objet!



cf. la discussion <http://stackoverflow.com/questions/10513633/static-method-access-to-non-static-constructor>

2.4. Utilité générale des enum

2.4.1. Modélisation

Le type enumération est souvent utilisé en modélisation :

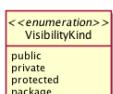


Figure 1. Exemple de classe Enumeration en UML

2.4.2. Propriétés

```
public enum Civilite {  
    MADAME, MONSIEUR  
}
```

- Chaque élément d'une énumération est un objet à part entière
- Les objets enum héritent de `java.lang.Enum`
- On peut compléter les comportements des objets en ajoutant des méthodes

2.4.3. Méthodes de base

- `toString()`

```
System.out.println(Civilite.MADAME); //MADAME
```

- `valueOf()`

```
Civilite civilite = Civilite.valueOf("MONSIEUR") ;
```

- `values()`

```
Civilite[] civilites = Civilite.values() ;
```

- `ordinal()`

```
Civilite civilite = Civilite.MONSIEUR ;
System.out.println("Civilite : " + civilite + " [" + civilite.ordinal() + "]") ;
// Civilite : MONSIEUR [1]
```



Le 1er numéro d'ordre est 0.

- `compareTo()`

```
System.out.println(Civilite.MADAME.compareTo(Civilite.MONSIEUR)) ;
// -1
```

2.4.4. Exemple plus complexe

(source : <http://openclassrooms.com/courses/apprenez-a-programmer-en-java/les-enumerations-1>)

```
public enum Langage {
    //Objets directement construits
    JAVA("Langage JAVA", "Eclipse"),
    C ("Langage C", "Code Block"),
    CPlus ("Langage C++", "Visual studio"),
    PHP ("Langage PHP", "PS Pad");

    private String name = "";
    private String editor = "";

    //Constructeur
    Langage(String name, String editor){
        this.name = name;
        this.editor = editor;
    }

    public void getEditor(){
        System.out.println("Editeur : " + editor);
    }

    public String toString(){
        return name;
    }

    public static void main(String args[]){
        Langage l1 = Langage.JAVA;
        Langage l2 = Langage.PHP;

        l1.getEditor();
        l2.getEditor();
    }
}
```

3. Construire ses applications

Pour générer un programme, une documentation, à partir des sources, on peut :

- Soit utiliser un environnement intégré comme **eclipse**
- Soit construire les sorties (on parle de **Build**) à partir des sources

Nous nous intéressons dans cette section à cette deuxième catégorie.

Il existe plusieurs outils :

- les scripts (shell Unix et .bat Windows)
- **make**

- [ant](#)
- [Maven](#)
- [Ivy](#)
- [Gradle](#)

3.1. Les scripts

Un script shell

```
#!/bin/sh
UML='model.uml'
TYPE='PNG'
MAINPATH='/Users/bruel/localdev/cpoa'
DOCLETPATH=$MAINPATH/doclet
PUMLPATH=$MAINPATH/util
echo "Creating $UML..."
echo $DOCLETPATH

javadoc \
-private \
-quiet \
-J-DdestinationFile=$UML \
-J-DcreatePackages=false \
-J-DshowPublicMethods=true \
-J-DshowPublicConstructors=false \
-J-DshowPublicFields=true \
-doclet de.mallox.doclet.PlantUMLDoclet -docletpath $DOCLETPATH/plantUmlDoclet.jar \
-sourcepath . src/Canard.java src/Colvert.java

echo "Done creating plantUML model"

TYPE='png'
echo "Converting $UML to $TYPE..."
java -jar $PUMLPATH/plantuml.jar \
-config $PUMLPATH/config.cfg \
-t $TYPE $UML
echo "Done generating PNG from model"
```

Un script batch Windows

```
set UML=TD1.uml
set TYPE='PNG'
set DOCLETPATH=E:\IUT-S3\CP0A\TP1\SuperCanardBof
echo "Creating %UML%..."
rem javadoc -private -quiet -J-DdestinationFile=%UML% -J-DcreatePackages=false -J
-DshowPublicMethods=true -J-DshowPublicConstructors=false -J-DshowPublicFields=true
-doclet de.mallox.doclet.PlantUMLDoclet -docletpath %DOCLETPATH%\plantUmlDoclet.jar
src\canard\*.java
echo "Done."

javadoc -J-DdestinationFile=%UML% -J-DcreatePackages=false -J-DshowPublicMethods=true
-J-DshowPublicConstructors=false -J-DshowPublicFields=true -doclet
de.mallox.doclet.PlantUMLDoclet -docletpath plantUmlDoclet.jar src\appli\*.java
src\armes\*.java src\armes\impl\*.java

set TYPE='png'
echo "Converting %UML% to $TYPE..."
java -jar %DOCLETPATH%\plantuml.jar -config "%DOCLETPATH%\config.cfg" -t %TYPE% %UML%
echo "Done."
```

Avantages

- Faciles
- Rapides
- Beaucoup d'exemples

Inconvénients

- Pas portables sur d'autres systèmes (no comment ;-)
- Peu lisibles
- Peu évolutifs

3.2. make

Un Makefile pour générer ces cours

```
#-----
ICONSDIR=images/icons
IMAGESDIR=images
STYLE=/Users/bruel/Dropbox/Public/dev/asciidoc/stylesheets/golo-jmb.css
DOCTOR=asciidoc -a icons -a iconsdir=$(ICONSDIR) -a images=$(IMAGESDIR) -a source-highlighter=$(HIGHLIGHT)
DECK=swiss
EXT=asc
PANDOC=pandoc
OUTPUT=.
DEP=definitions.txt glossaire.txt refs.txt
#-----

all: $(OUTPUT)/*.html

images/%.png: images/%.plantuml
@echo '==> Compiling plantUML files to generate PNG'
java -jar plantuml.jar $<

%.html: %.$(EXT) $(DEP)
@echo '==> Compiling asciidoc files with Asciidoctor to generate HTML'
$(DOCTOR) -a toc2 -b html5 -a numbered -a eleve $<

%.deckjs.html: %.$(EXT) $(DEP)
@echo '==> Compiling asciidoc files to generate Deckjs'
$(DOCTOR) -T /Users/bruel/dev/asciidoc-backends/haml/deckjs/ -a slides \
-a data-uri -a deckjs_theme=$(DECK) \
-a icons -a iconsdir=$(ICONSDIR) \
-a images=$(IMAGESDIR) -a prof -o $@ $<

%-sujet.html: %.$(EXT) $(DEP)
@echo '==> Compiling asciidoc files with Asciidoctor to generate HTML'
$(DOCTOR) -a compact -a theme=compact -b html5 -a numbered -a eleve \
-a data-uri $< -o $@

%-prof.html: %.$(EXT) $(DEP)
@echo '==> Compiling asciidoc files with Asciidoctor to generate HTML'
$(DOCTOR) -a prof -a correction -a theme=compact -b html5 -a numbered \
-a data-uri $< -o $@
```

```
%.html: %.$(EXT) $(DEP)
@echo '==> Compiling asciidoc files with Asciidoctor to generate HTML'
$(DOCTOR) -a toc2 -b html5 -a numbered -a eleve $<
```

Exemple d'utilisation :

```
$ make wip.html
==> Compiling asciidoc files with Asciidoctor to generate HTML
asciidoctor -a icons -a iconsdir=images/icons -a images=images -a source-
highlighter=pygments -a toc2 -b html5 -a numbered -a eleve wip.asc
...
$ make wip.html
make: 'wip.html' is up to date.
```

3.3. ant

build.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<project default="main" name="EssaiBuild">
    <target name="main">
        <echo message="Version d'Ant utilisée: ${ant.version}" />
        <javadoc doclet="de.mallox.doclet.PlantUMLDoclet"
            docletpath="plantUmlDoclet.jar"
            access="private"
            additionalparam=
                "-encoding utf-8 -J-DdestinationFile=uml.txt -J-DcreatePackages=false -J
                -DshowPublicMethods=true -J-DshowPublicConstructors=false -J-DshowPublicFields=true"
        >
            <packageset dir="../src">
                <include name="**"/>
            </packageset>
        </javadoc>

        <java jar="plantuml.jar" fork="true" maxmemory="128m">
            <arg value="uml.txt"/>
        </java>
    </target>
</project>
```

Exemple d'utilisation :

```

$ ant main
Buildfile: build.xml
main:
[javadoc] Generating Javadoc
[javadoc] Javadoc execution
[javadoc] Loading source files for package pizzeriafactorysample...
[javadoc] Constructing Javadoc information...
[javadoc] PlantUMLDoclet.createPlantUml() - start
[javadoc] open outputfile: uml.txt
[javadoc] write interfaces/ abstract classes...
[javadoc] write content...
...
[javadoc] skip association for Pizza --> java.lang.String
[javadoc] skip association for Pizza --> java.lang.String
[javadoc] skip association for Pizza --> java.lang.String
[javadoc] skip association for Pizza --> java.util.ArrayList
[javadoc] PlantUMLDoclet.createPlantUml() - end
BUILD SUCCESSFUL
Total time: 9 seconds

```

Exemple d'utilisation dans **eclipse** pour la génération de fichier du type **TD.uml** :

1. Créer un répertoire tools et mettre dedans :

- **Plantuml.jar**
- **Plantumldoclet.jar**
- **build.xml**

2. Faire un **[Click_Droit]** sur **build.xml** et choisir **Run_As > Ant Build ...**

Bien choisir celui avec les points de suspensions.



* 1 Ant Build

* 2 Ant Build...

X Q

External Tools Configurations...

3. Dans l'onglet **Environment** :

- Créer une nouvelle variable de nom **Path** et de valeur : le répertoire de la JDK (où se trouve **javadoc** ou **javadoc.exe**)

4. Exécuter **[Run]**



Faire Refresh dans le navigateur pour voir les 2 fichiers générés (**uml.txt** et **uml.png**).

5. Il suffit de faire un [Click_Droit] sur `build.xml` et choisir Run_As > Ant Build ... (SANS les 3 points de suspensions) pour relancer la génération du diagramme.

3.4. Maven



Maven est un outil Java.

Convention over Configuration

Exemples de conventions :

- le code source est supposé se trouver dans `${basedir}/src/main/java`
- les différentes ressources dans `${basedir}/src/main/resources`
- les tests dans `${basedir}/src/test`
- un projet est supposé produire un fichier JAR
- Maven suppose que vous voulez compiler en bytecode dans `${basedir}/target/classes`
- et ensuite créer votre fichier JAR distribuable dans `${basedir}/target`

`pom.xml`

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook</groupId>
  <artifactId>my-project</artifactId>
  <version>1.0</version>
</project>
```

La commande :

```
$ mvn install
```

- va traiter les ressources,
- compiler le source,
- exécuter les tests unitaires,
- créer un JAR, et
- installer ce JAR dans le dépôt local.

La commande :

```
$ mvn site
```

va créer un fichier `index.html` dans `target/site` contenant des liens vers la JavaDoc et quelques

rapports sur votre code source.

Pour comparer, voici l'équivalent ant :

```
<project name="my-project" default="dist" basedir=".">
  <description>
    simple example build file
  </description>
  <!-- set global properties for this build -->
  <property name="src" location="src/main/java"/>
  <property name="build" location="target/classes"/>
  <property name="dist" location="target"/>

  <target name="init">
    <!-- Create the time stamp -->
    <tstamp/>
    <!-- Create the build directory structure used by compile -->
    <mkdir dir="${build}" />
  </target>

  <target name="compile" depends="init"
         description="compile the source " >
    <!-- Compile the java code from ${src} into ${build} -->
    <javac srcdir="${src}" destdir="${build}" />
  </target>

  <target name="dist" depends="compile"
         description="generate the distribution" >
    <!-- Create the distribution directory -->
    <mkdir dir="${dist}/lib"/>

    <!-- Put everything in ${build} into the MyProject-${DSTAMP}.jar
        file -->
    <jar jarfile="${dist}/lib/MyProject-${DSTAMP}.jar"
        basedir="${build}" />
  </target>

  <target name="clean"
         description="clean up" >
    <!-- Delete the ${build} and ${dist} directory trees -->
    <delete dir="${build}" />
    <delete dir="${dist}" />
  </target>
</project>
```



3.5. Ivy

ivy.xml

```
<ivy-module version="2.0">
  <info organisation="org.apache" module="hello-ivy"/>
  <dependencies>
    <dependency org="commons-lang" name="commons-lang" rev="2.0"/>
    <dependency org="commons-cli" name="commons-cli" rev="1.0"/>
  </dependencies>
</ivy-module>
```

build.xml

```
<project xmlns:ivy="antlib:org.apache.ivy.ant" name="hello-ivy" default="run">

  ...
  <!-- =====
  target: resolve
  ===== -->
  <target name="resolve" description="--> retrieve dependencies with ivy">
    <ivy:retrieve />
  </target>
</project>
```

3.6. Graddle

Gradle combine la flexibilité de [ant](#) avec les conventions de [Maven](#) mais évite les inconvénients de [XML](#).

build.gradle

```
task hello {
  doLast {
    println 'Hello world!'
  }
}
```

```
$ gradle hello
:hello
Hello world!
```

```
BUILD SUCCESSFUL
```

```
Total time: 3.486 secs
```

4. Design patterns

4.1. Introduction : importance des patrons

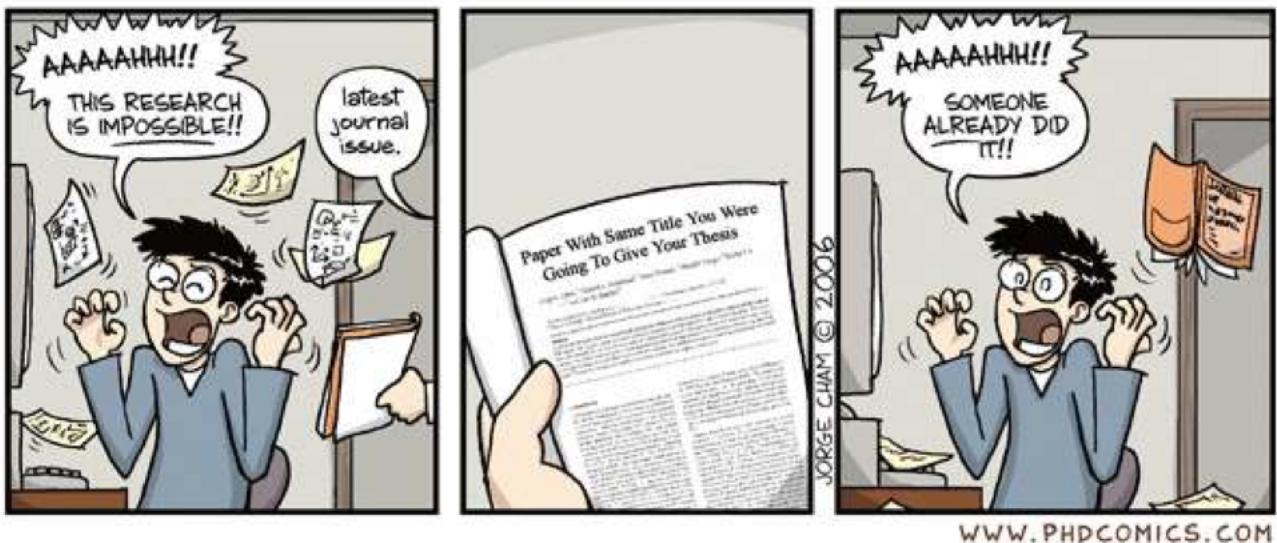


Figure 2. Les patrons : des réponses éprouvées à des problèmes récurrents

Science is what we understand well enough to explain to a computer. Art is everything else we do.

— Donald Knuth

4.1.1. Le patron *Strategy*

Principes de conception

Principe de conception



Identifiez les aspects de votre code qui varient et séparez-les de ceux qui demeurent constant.

Principe de conception



Programmer une interface, non une implémentation.

Principe de conception



Préférez la composition à l'héritage.

Définition du patron

Design pattern : Stratégie (Strategy)

Stratégie définit une famille d'algorithmes, encapsule chacun d'eux et les rend interchangeables. Il permet à l'algorithme de varier indépendamment des clients qui l'utilisent.

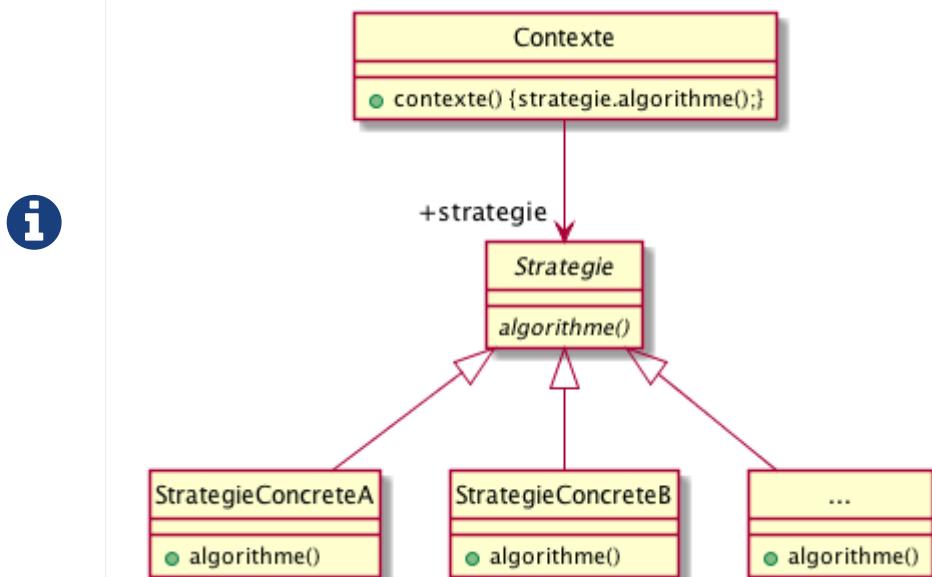


Figure 3. Modèle UML du patron Strategy

Premier exemple d'utilisation

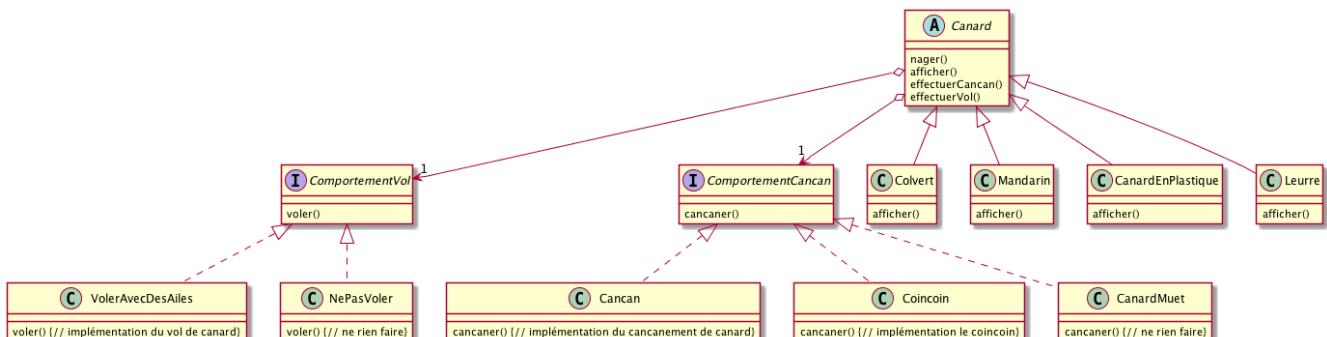


Figure 4. Premier exemple d'utilisation de patron



Pourquoi n'a-t-on pas utilisé *Strategy* pour `afficher()` ou `nager()`?

Autre exemple concret



L'exemple qui suit est tiré de [ce cours](#).

Le problème

Vous avez une classe **FileWriter** qui a pour rôle d'écrire dans un fichier ainsi qu'une classe **DBWriter**. Dans un premier temps, ces classes ne contiennent qu'une méthode `write()` qui n'écrira que le texte passé en paramètre.

Au fil du temps, vous vous rendez compte que c'est dommage qu'elles ne fassent que ça et vous aimerez bien qu'elles puissent écrire en différents formats (HTML, XML, etc.) : les classes doivent

donc formater puis écrire.

La solution

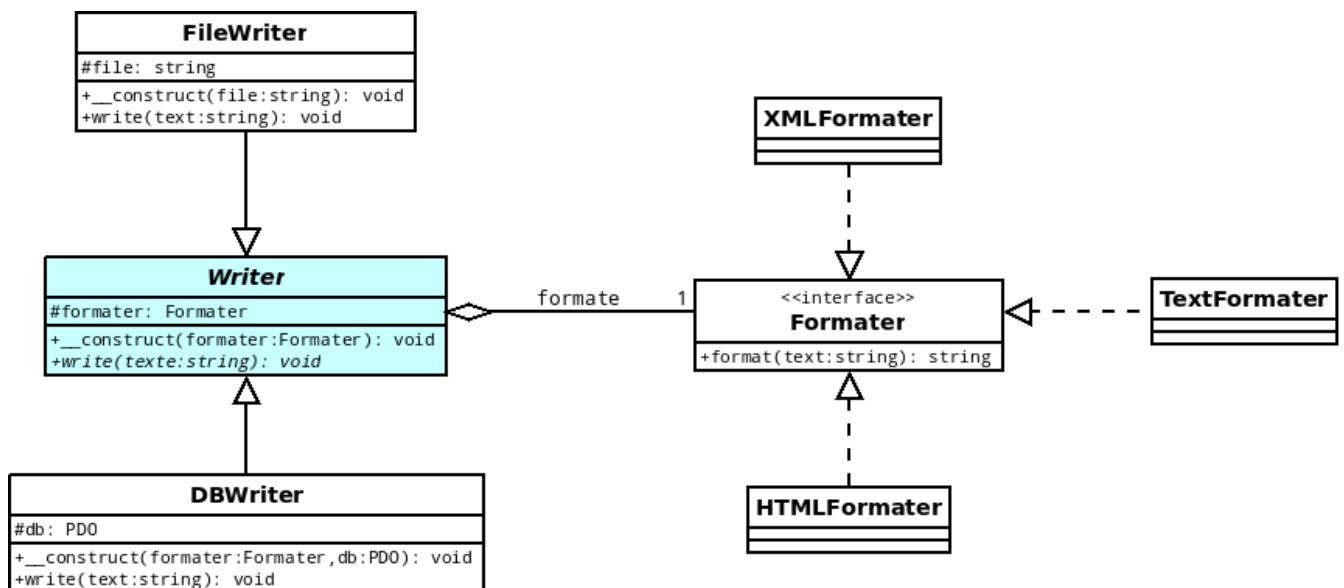


Figure 5. Application du pattern [strategy] (source)

L'interface en PHP (code source [ici](#))



```
<?php
interface Formater
{
    public function format($text);
}
?>
```

La classe abstraite Writer (code source [ici](#))



```
<?php
abstract class Writer
{
    // Attribut contenant l'instance du formateur que l'on veut
    // utiliser.
    protected $formater;

    abstract public function write($text);

    // Nous voulons une instance d'une classe implementant Formater en
    // paramètre.
    public function __construct(Formater $formater)
    {
        $this->formater = $formater;
    }
}
?>
```

La classe `FileWriter` ([code source ici](#))



```
<?php
class FileWriter extends Writer
{
    // Attribut stockant le chemin du fichier.
    protected $file;

    public function __construct(Formatter $formater, $file)
    {
        parent::__construct($formater);
        $this->file = $file;
    }

    public function write($text)
    {
        $f = fopen($this->file, 'w');
        fwrite($f, $this->formater->format($text));
        fclose($f);
    }
}
?>
```

La classe `DBWriter` ([code source ici](#))



```
<?php
class DBWriter extends Writer
{
    protected $db;

    public function __construct(Formatter $formater, PDO $db)
    {
        parent::__construct($formater);
        $this->db = $db;
    }

    public function write ($text)
    {
        $q = $this->db->prepare('INSERT INTO lorem_ipsum SET text =
:text');
        $q->bindValue(':text', $this->formater->format($text));
        $q->execute();
    }
}
?>
```

Enfin, nous avons nos trois formateurs. L'un ne fait rien de particulier (`TextFormatter`), et les deux autres formatent le texte en deux langages différents (`HTMLFormatter` et `XMLFormatter`).

La classe TextFormater (code source [ici](#))

```
<?php
class TextFormater implements Formater
{
    public function format($text)
    {
        return 'Date : ' . time() . "\n" . 'Texte : ' . $text;
    }
}
?>
```

La classe HTMLFormater (code source [ici](#))



```
<?php
class HTMLFormater implements Formater
{
    public function format($text)
    {
        return '<p>Date : ' . time() . '<br />' . "\n". 'Texte : ' . $text
. '</p>';
    }
}
?>
```

La classe XMLFormater (code source [ici](#))

```
<?php
class XMLFormater implements Formater
{
    public function format($text)
    {
        return '<?xml version="1.0" encoding="ISO-8859-1"?>' . "\n".
            '<message>' . "\n".
            "\t". '<date>' . time() . '</date>' . "\n".
            "\t". '<texte>' . $text . '</texte>' . "\n".
            '</message>';
    }
}
?>
```

D'autres exemples

- La fonction standard `sort()` de python

```
>>> sorted("This is a test string from Andrew".split(), key=str.lower)
['a', 'Andrew', 'from', 'is', 'string', 'test', 'This']
```

- Stratégie de cryptage en fonction de la taille d'un fichier

```

File file = getFile();
Cipher c = CipherFactory.getCipher( file.size() );
c.performAction();

// implementations:
interface Cipher {
    public void performAction();
}

class InMemoryCipherStrategy implements Cipher {
    public void performAction() {
        // load in byte[] ....
    }
}

class SwapToDiskCipher implements Cipher {
    public void performAction() {
        // swap partial results to file.
    }
}

```



Plus de détails [ici](#)

4.1.2. (non) Réutilisation



Les patrons **ne sont pas réutilisables!**

Il faut implémenter la solution qu'il représente à chaque fois.

Exception : certains font l'objet d'une librairie.

Par exemple le patron Singleton existe dans la bibliothèque standard du langage en [Ruby](#). C'est un mixin qu'il suffit d'inclure dans la classe qui doit être un singleton.

```

class Klass
  include Singleton
  # ...
end

a,b = Klass.instance, Klass.instance

a == b
# => true

Klass.new
# => NoMethodError - new is private ...

```

4.1.3. Association ou composition

On trouve deux modèles UML™ :

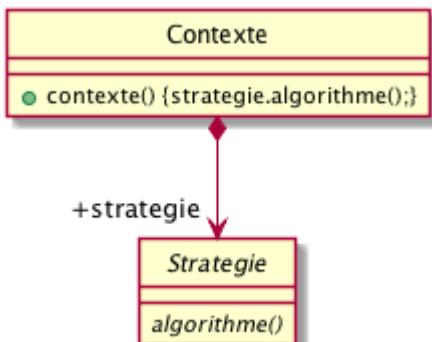


Figure 6. Strategy et composition

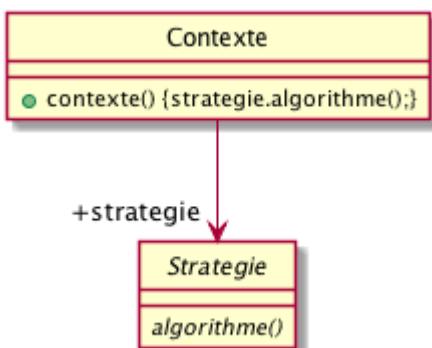


Figure 7. Strategy et association

Et donc deux implémentations :

Composition ⇒ le composé encapsule les composants

```
public class Colvert extends Canard {

    protected Colvert() {
        this(new VolerAvecDesAiles(), new Cancan());
    }

    ...
    c1 = new Colvert();
}
```

Association ⇒ le composant existe "en dehors"

```
...
vol = new VolerAvecDesAiles();
cri = new Cancan();
c1 = new Colvert(vol,cri);
...
```

4.2. Un peu d'histoire

1977

Alexander : patterns pour les architectures (les vraies) 

1987

Beck et Cunningham : patterns pour des interfaces utilisateurs

1988

Meyer : livre sur l'orienté objet (langage Eiffel), devenu la bible pour beaucoup de programmeurs (cf. [Meyer88])

1990-1995

Gamma, Helm, Johnson et Vlissides : LE livre de référence (cf. [GoF]) 

2003

Martin : principes SOLID (cf. [Martin03])

2004

Craig Larman décrit des modèles de conception : les Patterns GRASP (cf. [Larman05])



Les patterns de ce livre sont connus comme les Gof pour « Gang of Four ».

4.3. Exemples de bons principes

SOLID:

- *Single Responsibility Principle*
- *Open-Closed Principle*
- *Liskov Substitution Principle*
- *Interface Segregation Principle*
- *Dependency Inversion Principle*

4.3.1. Single Responsibility Principle



Single Responsibility Principle

Just because you *can* doesn't mean you *should*.

Responsabilité => Sujet à changement

4.3.2. Open-Closed Principle



Open-Closed Principle

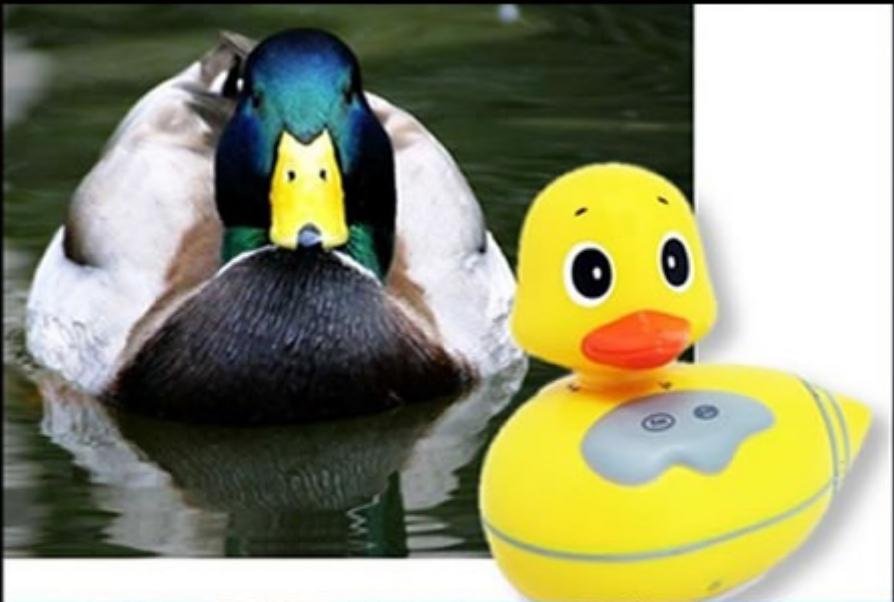
Open-chest surgery isn't needed when putting on a coat.

Ouvert à l'extension mais fermé à la modification



Une fois écrite et testée, une classe ne devrait être modifiée que pour être corrigée! Toute modification devrait être possible par extension.

4.3.3. Liskov Substitution Principle



Liskov Substitution Principle

If it looks like a duck and quacks like a duck but needs batteries,
you probably have the wrong abstraction.

Une classe doit pouvoir être remplacée par une instance d'un de ses sous-types, sans modifier la cohérence du programme

Un carré est un rectangle à deux côtés égaux.



Figure 8. Exemple classique de violation du principe de substitution de Liskov



Peut-on toujours substituer un Carré à la place d'un Rectangle ?

4.3.4. Interface Segregation Principle



Interface Segregation Principle

You want me to plug this in *where?*

Préférer plusieurs interfaces spécifiques pour chaque client plutôt qu'une seule interface générale

4.3.5. Dependency Inversion Principle



Dependency Inversion Principle

Would you solder a lamp directly
to the electrical wiring in a wall?

Il faut dépendre des abstractions, pas des implémentations

Ce principe indique :

- Les modules de haut niveau (abstraits) ne doivent pas dépendre des modules de bas niveau. Les deux doivent dépendre d'abstractions.
- Les abstractions ne doivent pas dépendre des détails d'implémentation. C'est l'inverse : les détails doivent dépendre des abstractions.



Ainsi ce principe va à l'encontre de l'intuition classique.

4.3.6. SOLID et patrons



QUESTION

Lesquels des 5 principes SOLID s'appliquent bien à *Strategy* ?

4.4. GRASP

The critical design tool for software development is a **mind well educated in design principles**. It is not the UML or any other technology.

— Craig Larman, 2005

Il s'agit d'un ensemble de patrons, plutôt orientés conception (UML). Nous en aborderons certains au travers des exemples de ce module (cf. [Larman05]).

4.5. Les patrons : comment ça marche ?

4.5.1. Intérêt

- Réponses éprouvées à des problèmes récurrents
- Vocabulaire commun

T'as qu'à utiliser une *factory*!

4.5.2. Définition

- Nom
- Problème
- Solution
- Conséquences

Exemple de *Strategy* :

Nom

Strategy

Problème

Situations où il est nécessaire de pouvoir définir dynamiquement les algorithmes utilisés.

Solution

Définir une famille d'algorithmes, encapsuler chacun d'eux en tant qu'objet, et les rendre interchangeables.

Conséquences

Ce patron laisse les algorithmes changer indépendamment des clients qui les emploient.

4.5.3. Patrons à aborder

- Ceux déjà pratiqués
 - [Singleton]
 - [Observateur]
 - [Fabrique] (*factory*) (cf. parser sax)
- Les "pressentis"
 - [Stratégie]
 - [Itérateur]
 - [Composite]

- [Etat]
- [Proxy]
- Les nouveaux
 - Décorateur
 - Commande
 - [Adaptateur]
 - Façade
 - Patron de méthode
- Les "avancés"
 - Chaînes de responsabilité
 - Visiteur
- Ceux qu'on n'aura pas le temps d'aborder
 - Prototype
 - Memento
 - Médiateur
 - Interprète
 - Poids-mouche
 - Monteur
 - Pont
- Concepts avancés
 - Patrons de patrons (exemple du MVC)
 - Anti-patrons

5. Le patron Fabrique

5.1. Principes de conception

Design pattern : Fabrique (Factory)

Fabrique (simple) définit une interface pour la création d'un objet, mais en laissant à des sous-classes le choix des classes à instancier (voir aussi [Fabrique abstraite](#)).

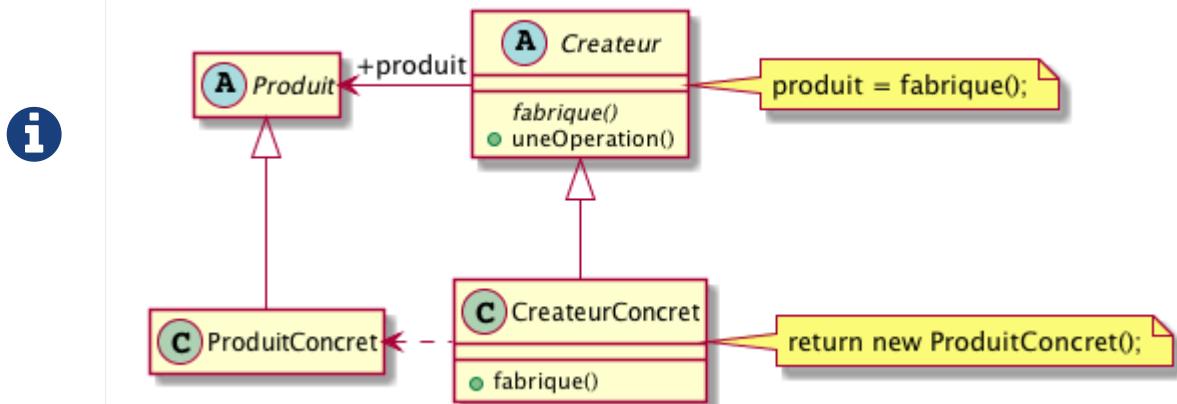


Figure 9. Modèle UML du patron Fabrique

5.2. Premier exemple d'utilisation de patron

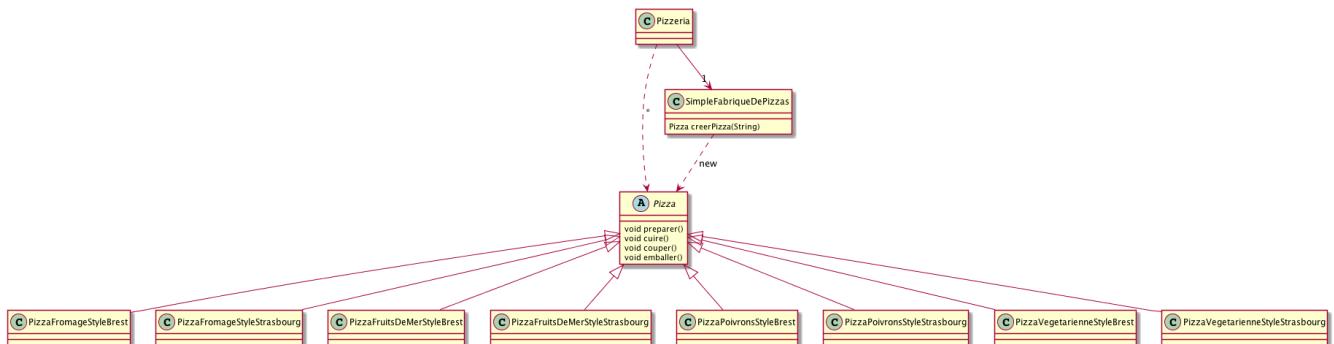


Figure 10. 1er exemple d'utilisation de Fabrique

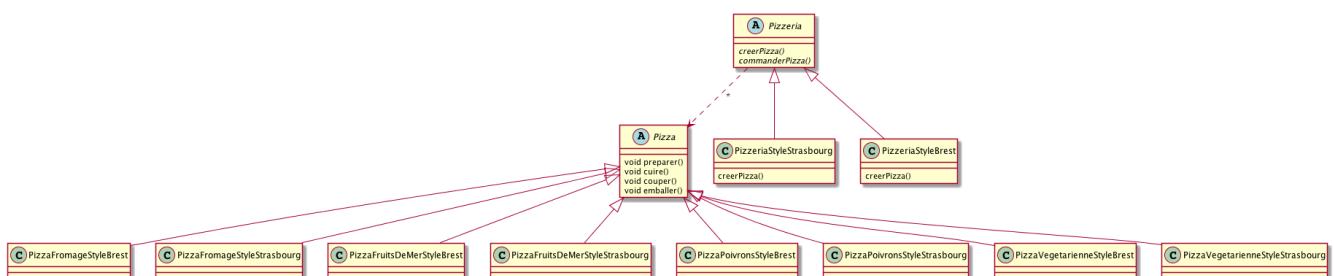


Figure 11. Exemple complet d'utilisation de Fabrique

5.3. Autre exemple concret

Factory en PHP (source [ici](#))

```
<?php
class DBFactory
{
    public static function load($sgbdr)
    {
        $classe = 'SGBDR_' . $sgbdr;

        if (file_exists($chemin = $classe . '.class.php'))
        {
            require $chemin;
            return new $classe;
        }
        else
        {
            throw new RuntimeException('La classe <strong>' . $classe . '</strong> n\'a pu
être trouvée !');
        }
    }
?>
```

Factory en PHP (source [ici](#))

```
<?php
try
{
    $mysql = DBFactory::load('MySQL');
}
catch (RuntimeException $e)
{
    echo $e->getMessage();
}
?>
```

Factory en java (source : votre module AA!)

```
public class XmlExprParser {  
  
    public static Expression fromFile(String file) throws ... {  
        SAXParserFactory spf = SAXParserFactory.newInstance();  
        spf.setValidating(true);  
        SAXParser sp = spf.newSAXParser();  
        ExprHandler ep = new ExprHandler();  
        sp.parse(file, ep);  
        return ep.getResult();  
    }  
}
```

5.4. Mais c'est pas fini!

Reprendons nos pizzas vues en TD

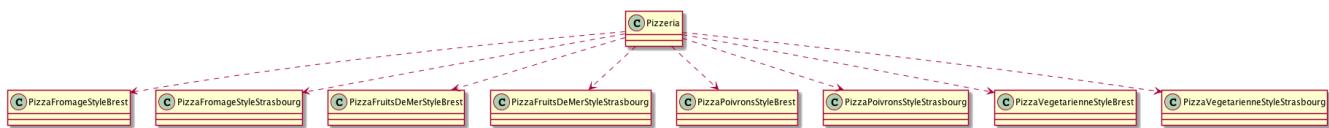


Figure 12. Une pizzeria dépendante

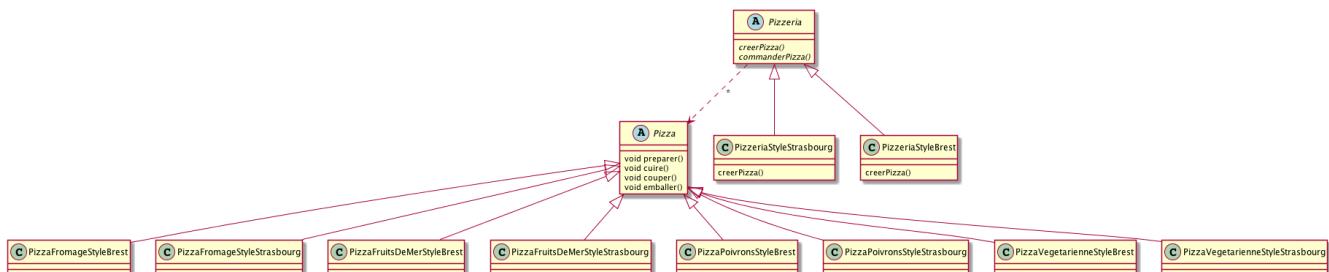


Figure 13. Une pizzeria dépendante

- Aucune variable ne doit contenir une référence à une classe concrète.
- Aucune classe ne doit dériver d'une classe concrète.
- Aucune classe ne doit redéfinir une méthode implémentée dans une classe de base.

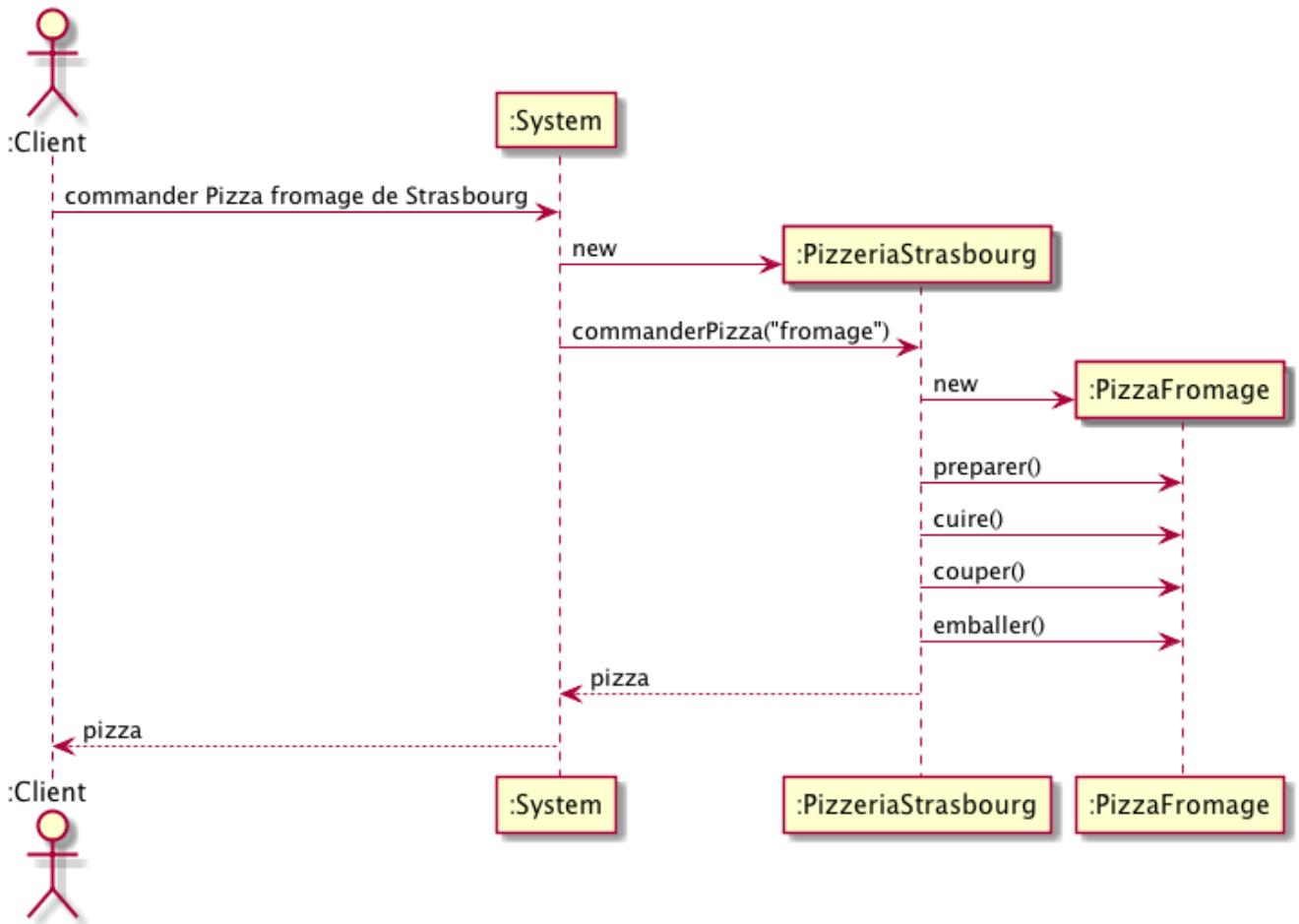


Figure 14. Une pizzeria avec Fabrique

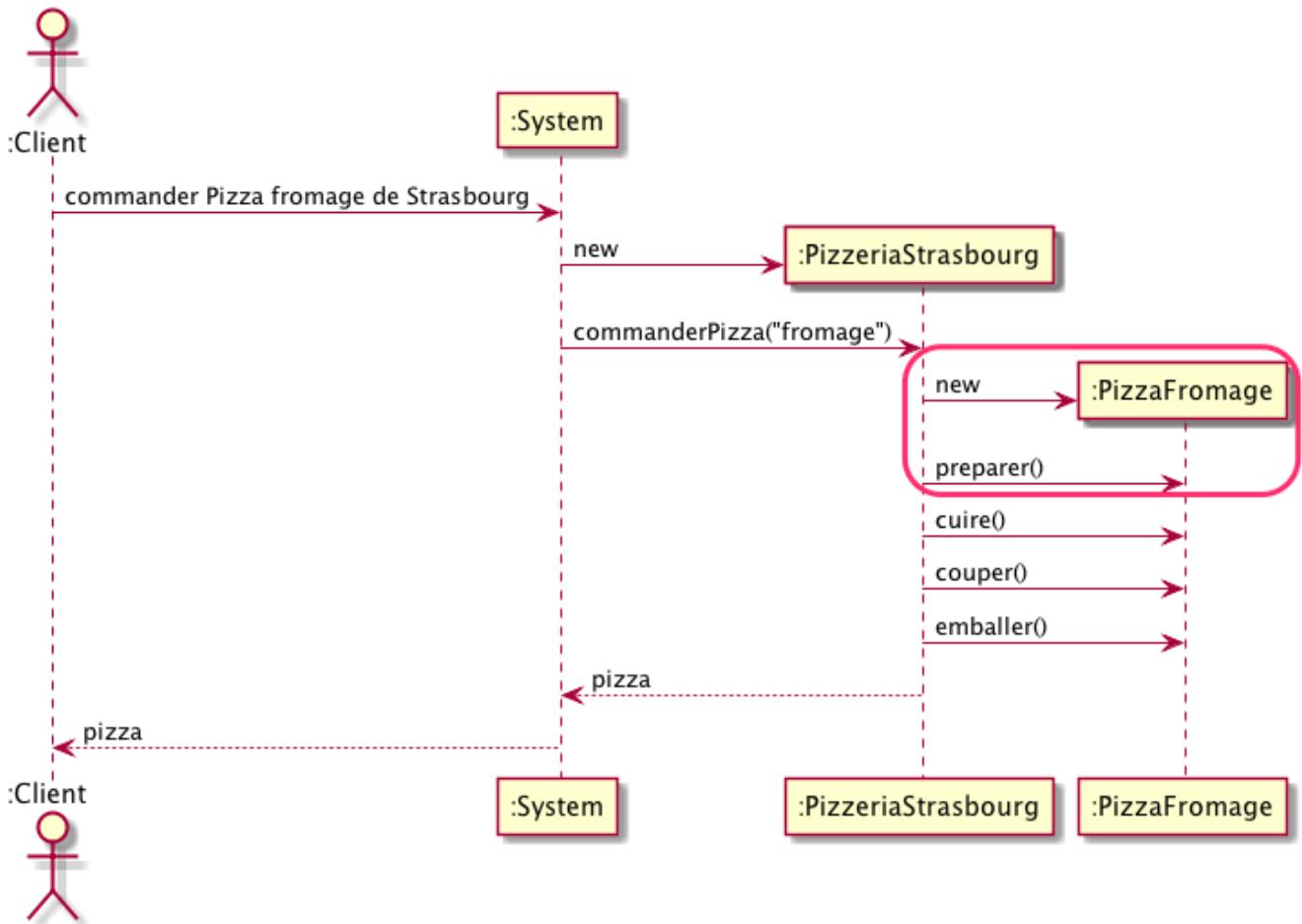


Figure 15. Problème de la dépendance des ingrédients



Figure 16. Des cartes adaptées

```
public interface FabriqueIngredientsPizza {  
    public Pate creerPate();  
    public Sauce creerSauce();  
    public Fromage creerFromage();  
    public Legumes[] creerLegumes();  
    public Poivrons creerPoivrons();  
    public Moules creerMoules();  
}
```

```
public class FabriqueIngredientsPizzaBrest implements FabriqueIngredientsPizza {  
    public Pate creerPate() {  
        return new PateFine();  
    }  
    public Sauce creerSauce() {  
        return new SauceMarinara();  
    }  
    ...  
}
```

```
public class FabriqueIngredientsPizzaStrasbourg implements FabriqueIngredientsPizza {  
    public Pate creerPate() {  
        return new PateEpaisse();  
    }  
    public Sauce creerSauce() {  
        return new SauceTomateCerise();  
    }  
    ...  
}
```

```
public class PizzaFromage extends Pizza {  
    FabriqueIngredientsPizza fabriqueIngredients;  
  
    public PizzaFromage(FabriqueIngredientsPizza fabriqueIngredients) {  
        this.fabriqueIngredients = fabriqueIngredients;  
    }  
    void preparer() {  
        System.out.println("Preparation de " + nom);  
        pate = fabriqueIngredients.creerPate();  
        sauce = fabriqueIngredients.creerSauce();  
        fromage = fabriqueIngredients.creerFromage();  
    }  
}
```

5.5. Fabrique abstraite

Nous sommes arrivé à une version du patron Fabrique appelée **Fabrique Abstraite** :

Fabrique (abstraite) fournit une interface pour la création de familles d'objets apparentés ou interdépendants, sans qu'il soit nécessaire de spécifier leurs classes concrètes (voir aussi [Fabrique](#)).

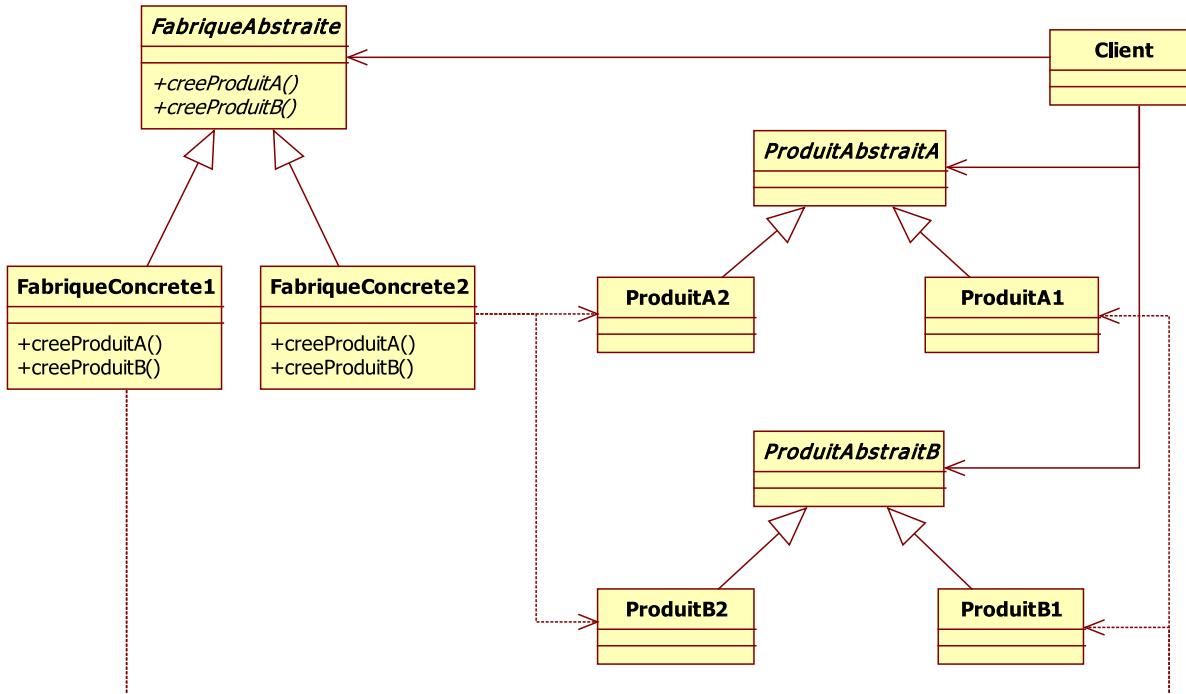


Figure 17. Modèle UML du patron Fabrique Abstraite

6. Un nouveau diagramme UML très utile

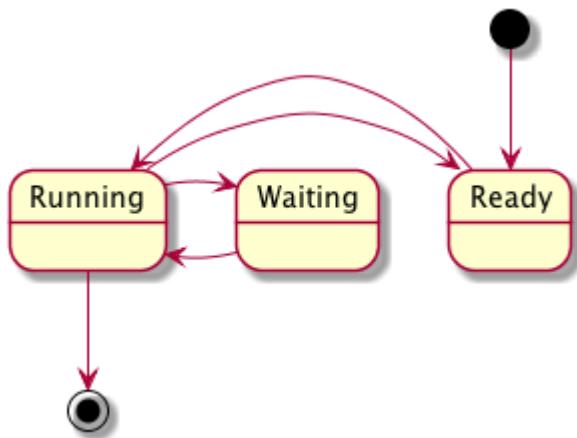


Figure 18. Exemple de diagramme d'état

Les diagrammes d'**états-transitions** (plus simplement diagramme d'état) d'**UML™** décrivent le comportement interne d'un objet à l'aide d'un automate à états finis.

Les notions importantes de ce diagramme :

- états

- actions
- événements déclencheurs
 - signaux
 - invocations de méthode

6.1. Transitions

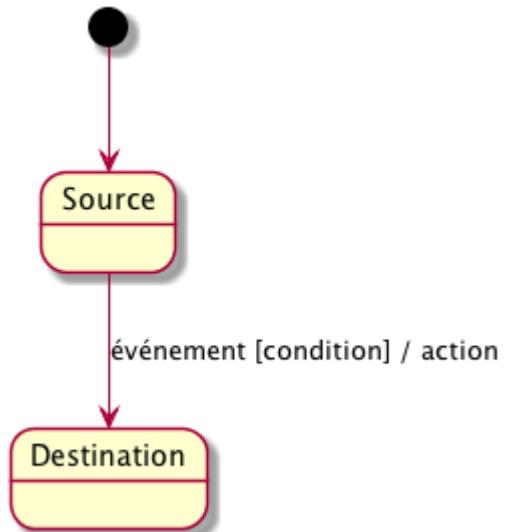


Figure 19. Transition entre états

Événement

Un signal, une invocation de méthode, etc.

Condition

Un booléen

Action

Affectation, invocation de méthode

6.2. Exemple de transitions

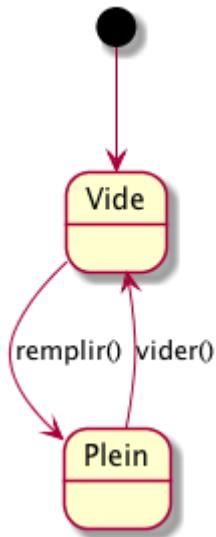


Figure 20. Transition entre états

6.3. Refactoring

On peut remplacer les actions systématiques des transitions entrantes :

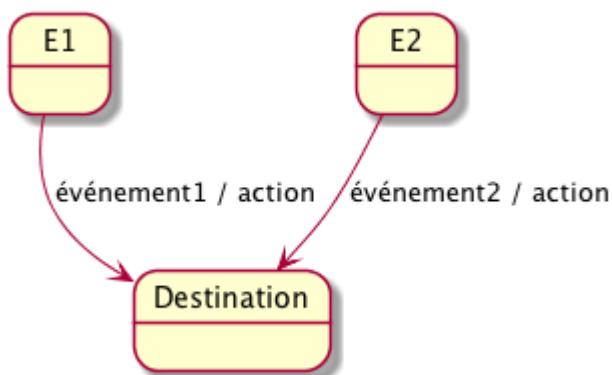


Figure 21. Transition entre états

par une **transition interne** : `entry` :

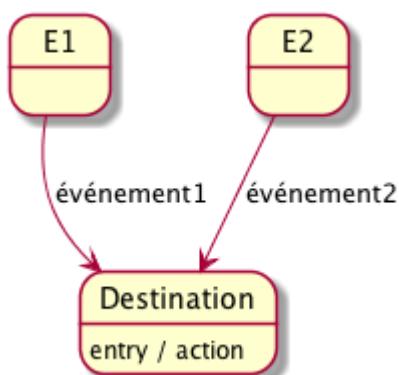


Figure 22. Exemple de transition interne

6.4. Transitions internes

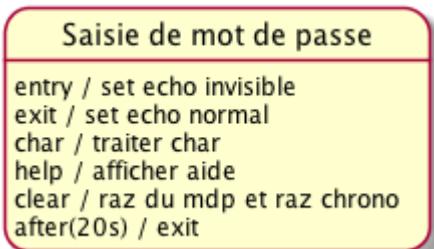


Figure 23. Exemple de transitions internes

entry

permet de spécifier une activité qui s'accomplit quand on entre dans l'état.

exit

permet de spécifier une activité qui s'accomplit quand on sort de l'état.

do

commence dès que l'activité **entry** est terminée. Lorsque cette activité est terminée, une transition d'achèvement peut être déclenchée. Si une transition se déclenche pendant que l'activité **do** est en cours, cette dernière est interrompue et l'activité **exit** de l'état s'exécute.

6.5. Conditions

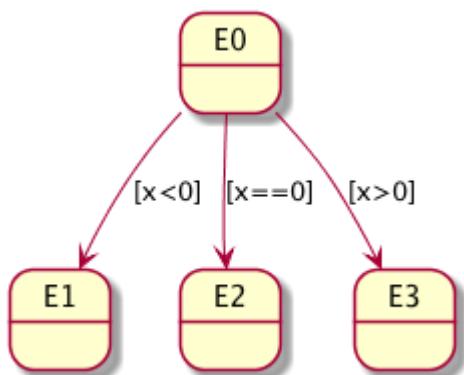


Figure 24. Les conditions doivent être exclusives

QUESTION



1. Réalisez un diagramme d'état UML représentant les différents états de l'eau (liquide, solide, gazeux).
2. Réalisez un diagramme d'état UML représentant les états d'un étudiant de son arrivée en 1ère année à sa sortie de l'IUT en fonction des résultats aux différents examens (uniquement les années, pas les semestres).

6.6. Etats complexes

Un état peut lui-même être doté d'un comportement et donc représenter à lui seul une machine à état. Par exemple :

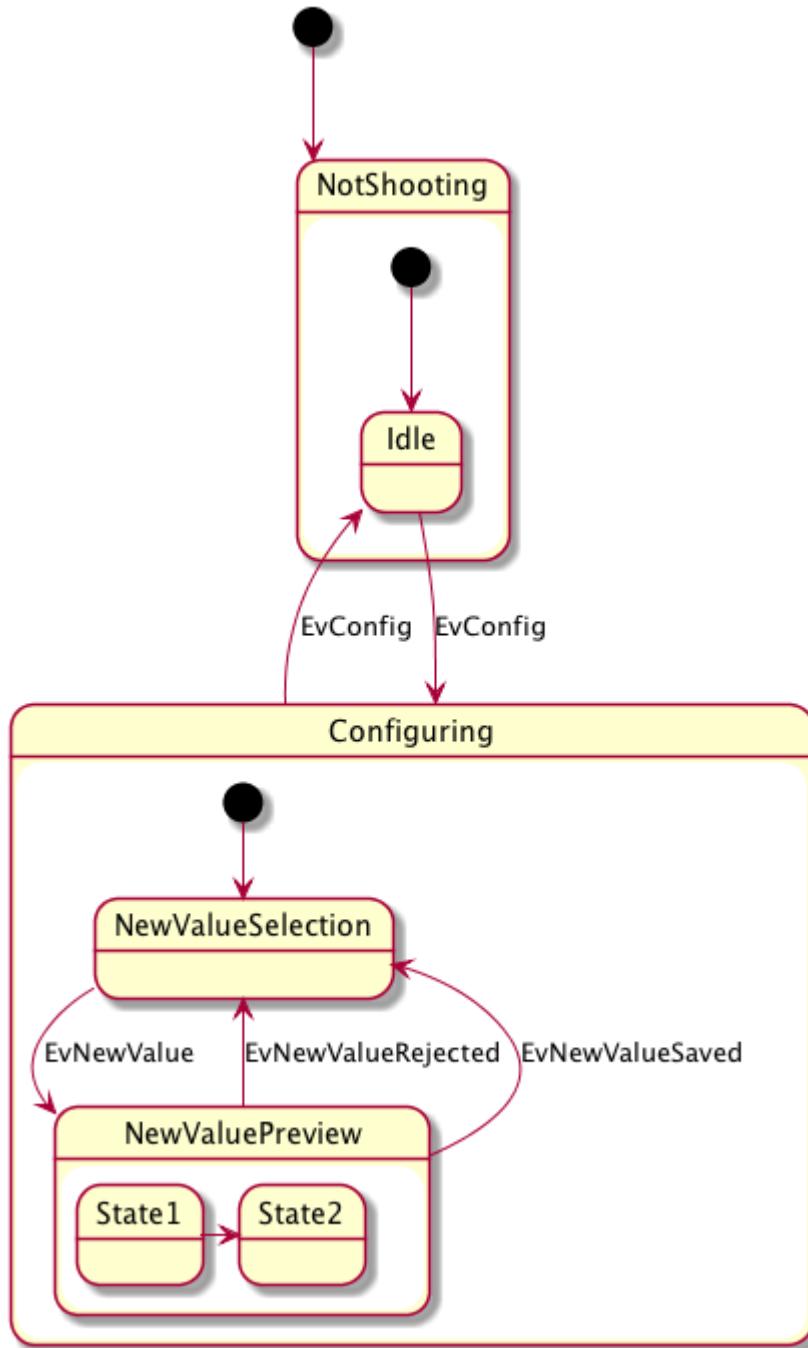


Figure 25. Etats "Composite"



QUESTION

Intégrez les semestres aux diagramme précédent (étudiants)

6.7. Notion de concurrence

On peut représenter l'évolution de différentes machines de manière concurrente (parallèle). Par exemple :

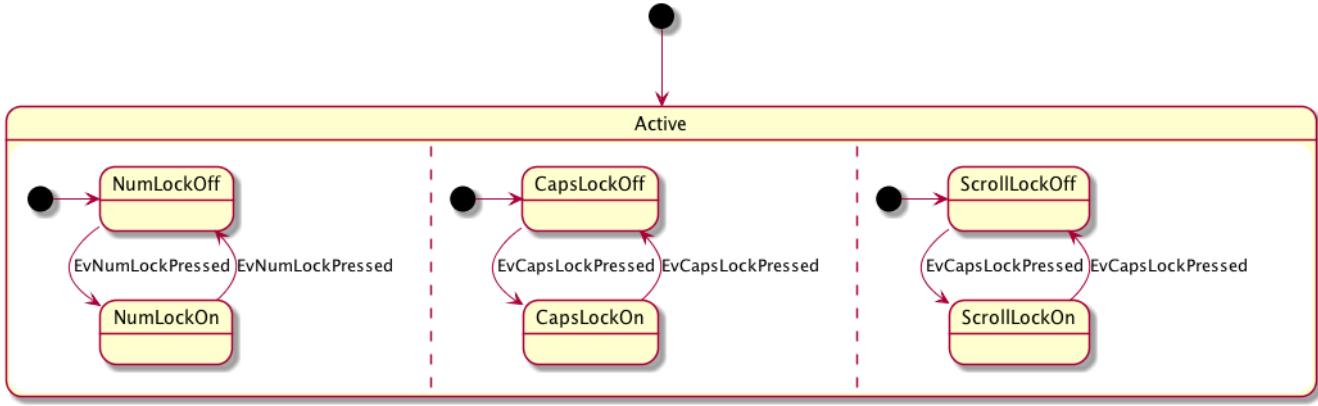


Figure 26. Etats "Concurrents"



QUESTION

Réalisez le diagramme d'état d'une machine à boisson rendant la monnaie.



Pour aller plus loin sur ce diagramme : <http://laurent-audibert.developpez.com/Cours-UML/?page=diagramme-etats-transitions>

7. Le patron Etat

Soit la machine à état suivante :

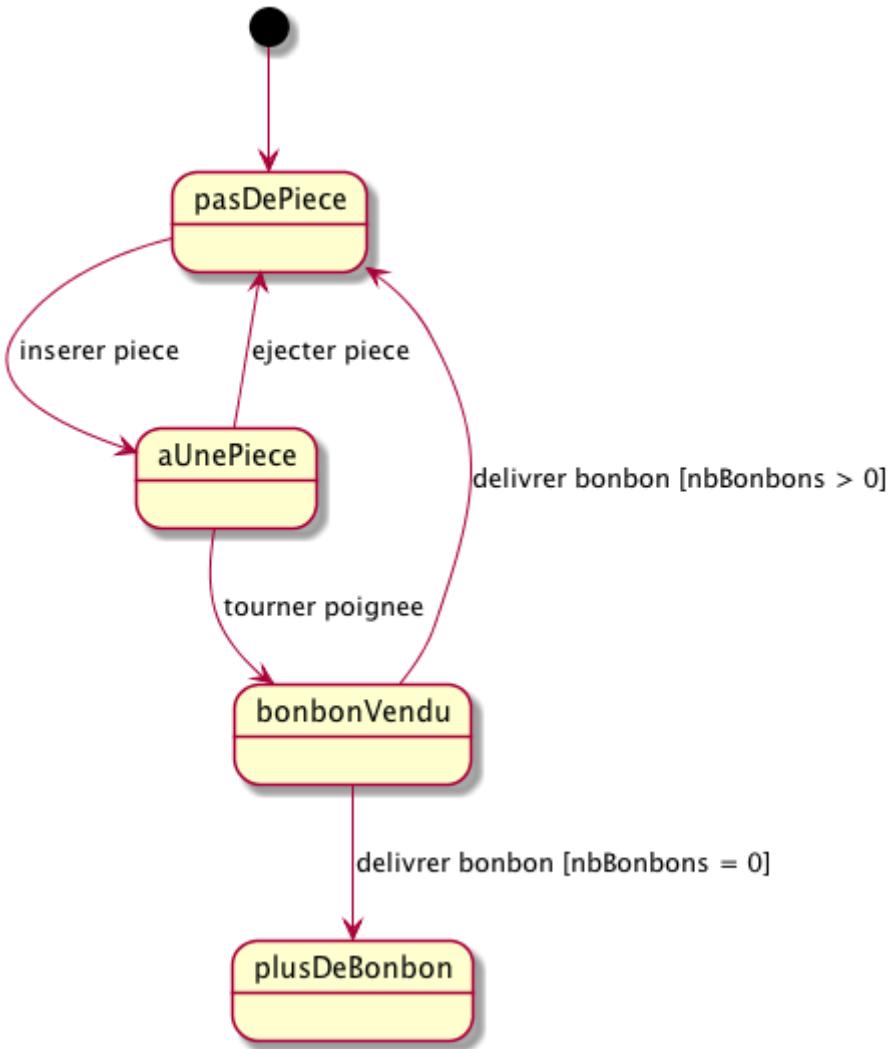


Figure 27. Machine à état d'un distributeur simple

7.1. Implémentation intuitive

Implémentation sans switch case

```

public void insererPiece() {
    if (etat == A_PIECE) {
        System.out.println("Vous ne pouvez plus insérer de pièces");
    } else if (etat == EPUISE) {
        System.out.println("Vous ne pouvez pas insérer de pièce, nous sommes en rupture de stock");
    } else if (etat == VENDU) {
        System.out.println("Veuillez patienter, le bonbon va tomber");
    } else if (etat == SANS_PIECE) {
        etat = A_PIECE;
        System.out.println("Vous avez inséré une pièce");
    }
}

```

7.2. Erreur d'implémentations

- Ce code n'adhère pas au principe Ouvert-Fermé.
- Cette conception n'est pas orientée objet.
- Les transitions ne sont pas explicites. Elles sont enfouies au milieu d'un tas d'instructions conditionnelles.
- Nous n'avons pas encapsulé ce qui varie.
- Les ajouts ultérieurs sont susceptibles de provoquer des bugs dans le code.

7.3. Une meilleure implémentation

1. Définir une nouvelle interface **Etat** qui contiendra une méthode pour chaque action
2. Implémenter une classe pour chaque **Etat**. Elles seront responsable du comportement.
3. Se débarrasser de toutes les instructions conditionnelles et les remplacer par une délégation à la classe adéquate.

7.4. Illustration

Etape 1 :

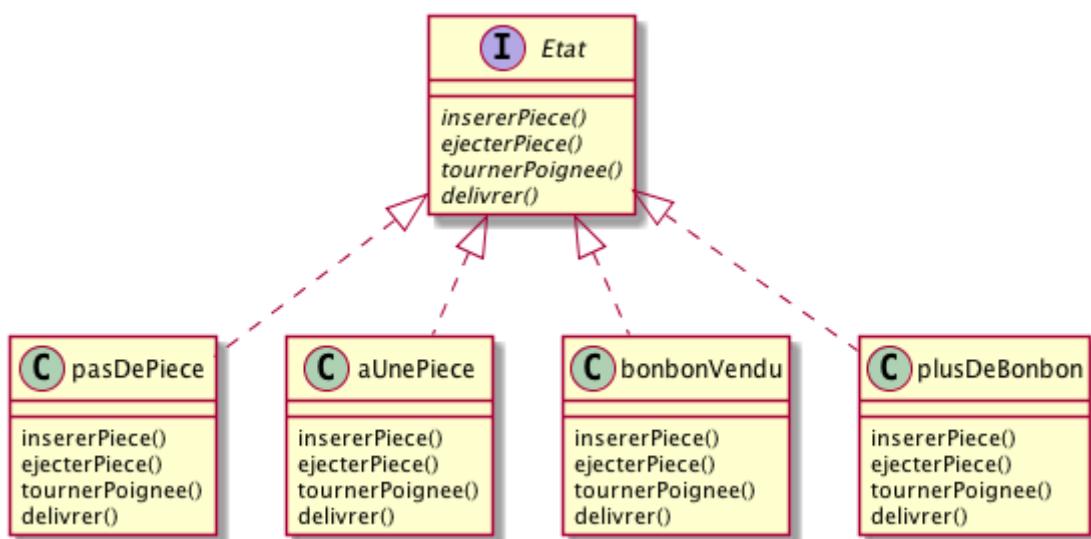
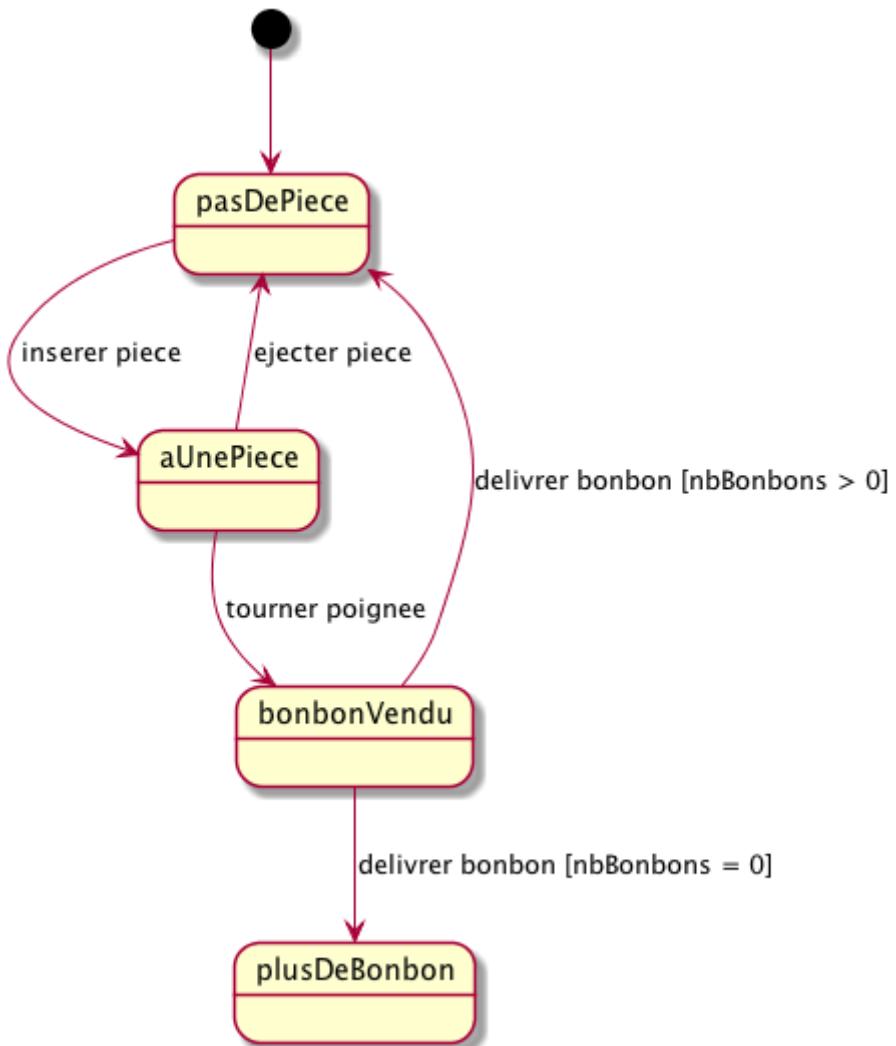


Figure 28. Implémentation des états

Etape 2

```

public class EtatSansPiece implements Etat {
    // Va falloir remplir ici...

    public void insererPiece() {
        System.out.println("Vous avez inséré une pièce");
        // changer d'état si besoin
    }
    ...
}

```

Etape 3

```

public class Distributeur {

    Etat etat = new EtatSansPiece(); // état initial
    ...

    public void insererPiece() {
        etat.insererPiece(); // on délègue à l'état le soin de réagir
    }
    ...
}

```

Etape 4 (enfin, retour sur l'étape 2) : une solution possible...

```

public class EtatSansPiece implements Etat {
    Distributeur distributeur; // référence au distributeur qu'on gère

    public EtatSansPiece(Distributeur distributeur) {
        this.distributeur = distributeur;
    }

    public void insererPiece() {
        System.out.println("Vous avez inséré une pièce");
        distributeur.setEtat(distributeur.getEtatAPiece());
    }
    ...
}

```

7.5. Le patron Etat

Etat permet à un objet de modifier son comportement, quand son état interne change. Tout se passe comme si l'objet changeait de classe.

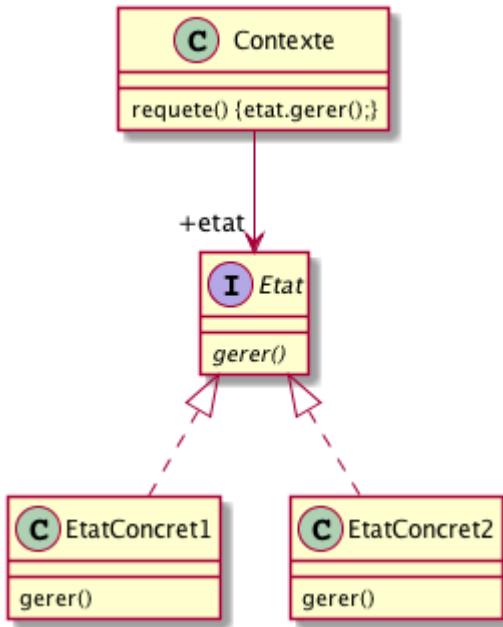


Figure 29. Modèle UML du patron Etat



QUESTION

Que pensez-vous de notre solution précédente par rapport à ce diagramme UML?

L'état possède une référence vers le contexte (**Distributeur** dans notre exemple).

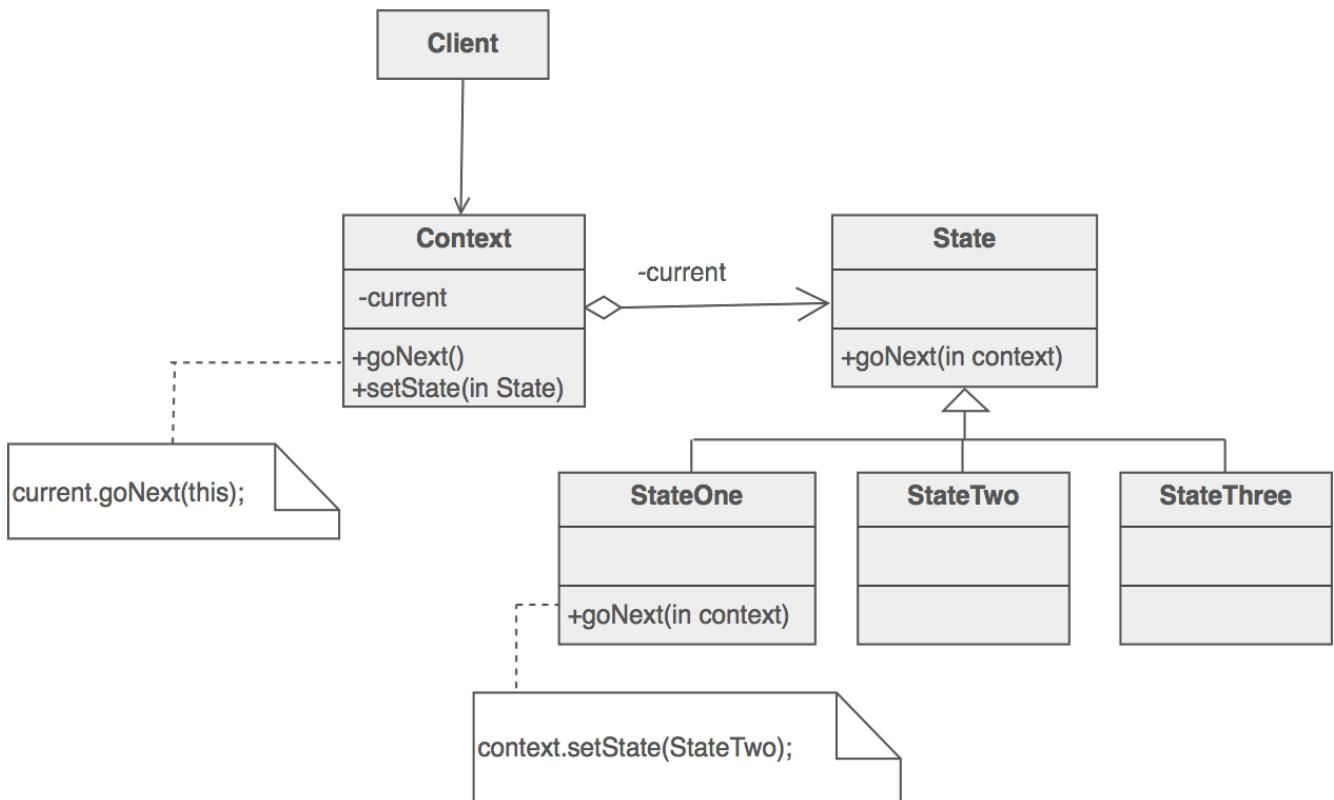


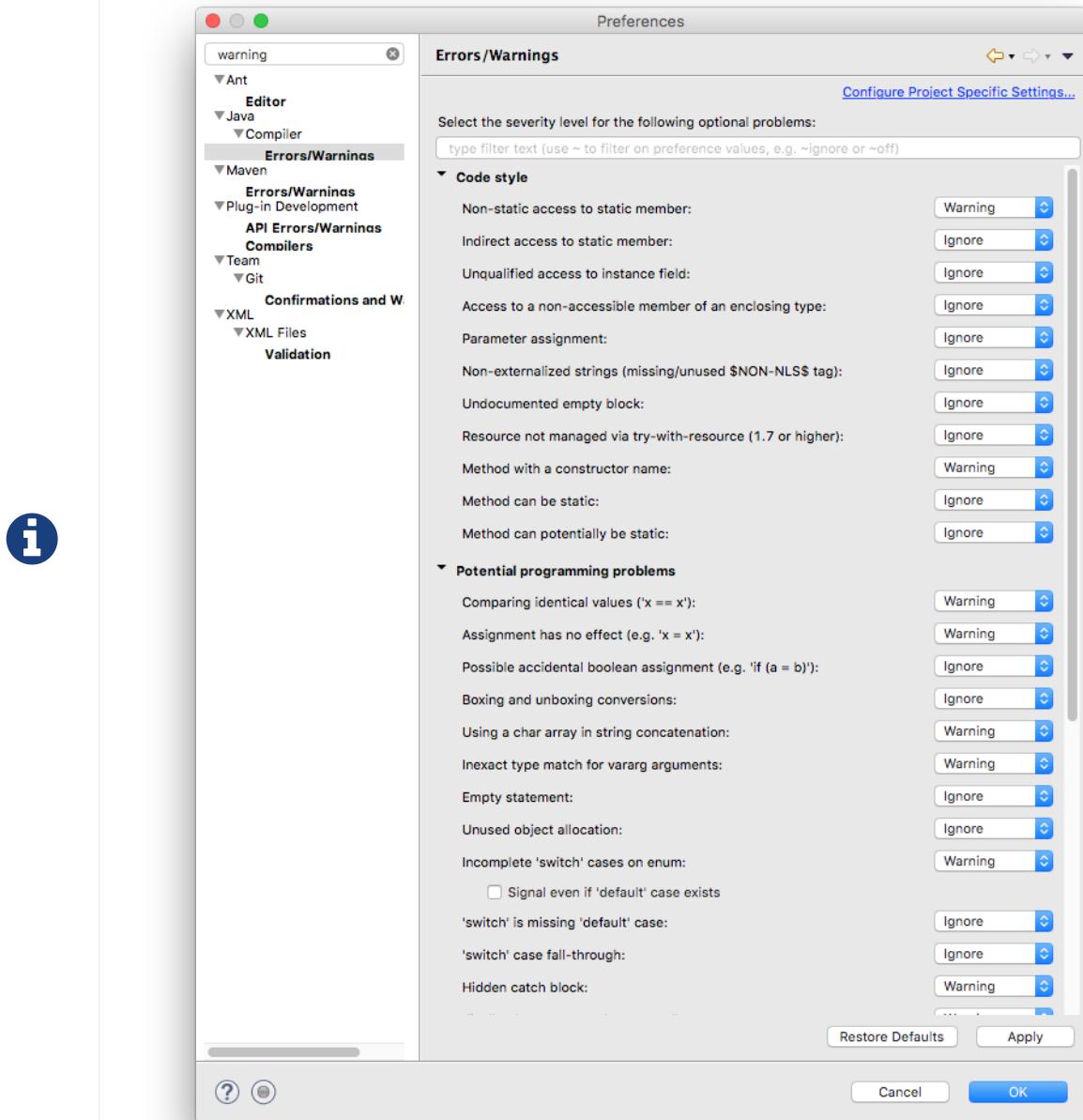
Figure 30. Une autre implémentation (source : https://sourcemaking.com/design_patterns/state)

8. Qualité du code

8.1. À minima : option de votre EDI

Les éditeurs permettent souvent de régler le niveau de détail des informations proposées.

Par exemple, dans [eclipse](#), on peut modifier le niveau de warning: **Preferences > Java > Compiler > Errors/Warnings**



8.2. Outils de base

Il existe des outils dédiés :

- [Findbugs](#)
- [PMD](#)
- [Checkstyle](#)

- Jacoco

8.3. Sonar

L'outil [SonarQube](#) permet de "mesurer" les problèmes de qualité en **dette technique** (le temps requis pour remédier à la faible qualité)

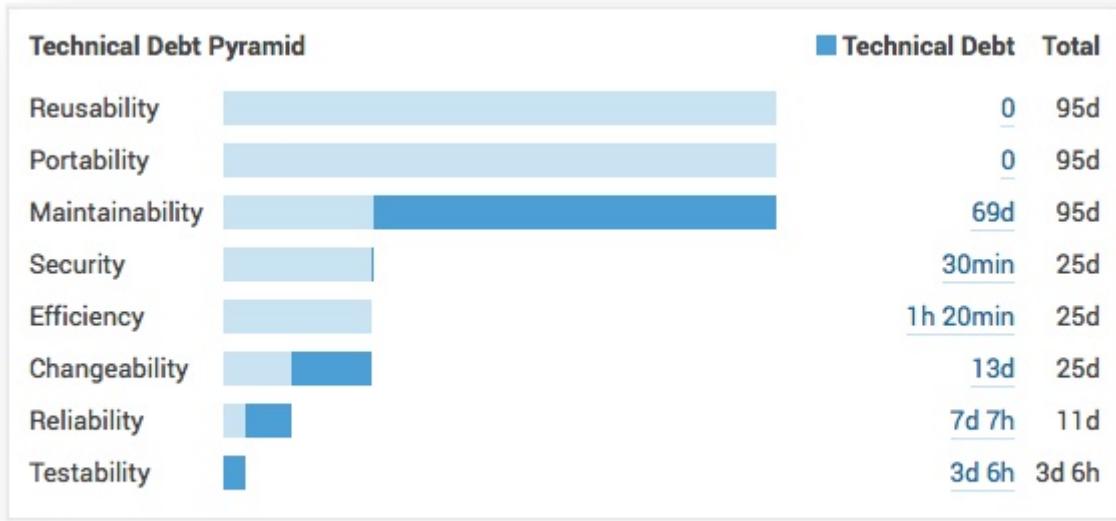


Figure 31. Dette technique (source <http://docs.sonarqube.org/display/HOME/Developers'+Seven+Deadly+Sins>)

Les aspects qui sont surtout considérés :

- Bugs (réels ou potentiels)
- Violation des standards de codage
- Duplications de code
- Manque de tests unitaires
- Mauvaise distribution de la complexité
- Code Spaghetti
- Pas assez (ou trop) de commentaires

9. Coding Dojo

9.1. Rappel des principes



Pour plus d'info sur le Coding Dojo : http://fr.wikipedia.org/wiki/Coding_Dojo

9.1.1. Le principe :

1. Le prof de TP (ou le/la gentil(le) étudiant(e) volontaire désintéressé(e)) initie l'exercice en ouvrant un **eclipse** et en commençant un test qui commence par échouer et qui "passera" quand la fonctionnalité attendue sera implémentée et fonctionnelle.
2. Il vérifie que son test échoue bien



On pourra avantageusement installer le plug-in eclipse **InfiniTest** qui permet de runner les tests à chaque sauvegarde (cf. plus haut)

3. Il fait en sorte que le test passe au vert le plus vite possible
4. Il "refactor" éventuellement (faire passer le test au vert mais de façon intelligente, éviter la duplication de code, réorganiser les classes, etc.)
5. Quand tout est au vert on passe à une fonctionnalité suivante

9.1.2. Les règles

- Toutes les 5 minutes (timer sonore) on change de personne aux commandes (au clavier/souris en l'occurrence)
- Il faut un "maître du temps"
- Tout le monde participe au codage (et ne fait pas du code de son côté)



Les machines ne sont donc pas utiles pour cette partie! Seule la machine connectée au vidéoprojecteur est utilisée.

- Le volontariat c'est mieux, mais le prof est libre d'organiser le tour de rôle.
- En 90' de TP on peut faire passer 18 personnes, donc tout le monde y passe et le plus tôt est en général le moins compliqué.
- Merci à ceux qui ont déjà participé aux coding dojos de montrer l'exemple ;-)

9.2. Sujet du jour : Type Abstrait Pile

Rappels

Opérations

```
CréerPile : -> Pile
estVide   : Pile -> Booléen
Empiler    : Pile * Elément -> Pile
Dépiler    : Pile -> Pile
Sommet     : Pile -> Elément
```

Préconditions



Sommet(p) valide Si et Seulement Si $\text{estVide}(p) == \text{FAUX}$
Dépiler(p) valide Si et Seulement Si $\text{estVide}(p) == \text{FAUX}$

Axiomes

- (1) $\text{estVide}(\text{CréerPile}())$
- (2) $\text{estVide}(\text{Empiler}(p, e)) == \text{FAUX}$
- (3) $\text{estVide}(\text{Dépiler}(\text{Empiler}(p, e)))$ Si et Seulement Si $\text{estVide}(p)$
- (4) $\text{Sommet}(\text{Empiler}(p, e)) == e$
- (5) $\neg \text{estVide}(p) \Rightarrow \text{Sommet}(\text{Dépiler}(\text{Empiler}(p, e))) == \text{Sommet}(p)$

L'axiome suivant résume à lui seul les axiomes (3) et (5) :

Dépiler($\text{Empiler}(p, e)$) == p

Choix d'implémentation

Le type abstrait Pile sera implémenté en **java** en utilisant un tableau de chaînes de caractères.



Dans un premier temps, la taille du tableau sera de 10 éléments.

Conséquences

- empiler() ajoutera une chaîne en sommet de pile et sommet() retournera une chaîne
- la pile pourra être pleine

Pour l'architecture logicielle, le type abstrait Pile sera composé :

1. d'une classe Pile fournissant un enregistrement de type Pile
2. d'une classe PileOperations fournissant toutes les opérations du type Abstrait Pile



Les étapes qui suivent ne sont pas complètement adaptées au dojo car souvent les tests se font après coup, mais vous pourrez avoir le code sous la main pour lancer les inspirations si elles ne viennent pas.

9.3. Étape 1 : Mise en place d'un enregistrement de type Pile

1. Commencer par un test simple et le placer dans un fichier de nom **PileTest.java**. Essayez de l'avoir en tête et de ne juste "ouvrir" un fichier déjà prêt. L'intérêt du coding dojo, c'est de voir comment vous utiliser les raccourcis, les quickfix etc.

```
import junit.textui.TestRunner;
import junit.framework.TestSuite;
import junit.framework.TestCase;

public class PileTest extends TestCase {
    static int totalAssertions = 0;
    static int bilanAssertions = 0;

    /*
     Opérations du type Pile
    */
    public void test_type_new_Pile() throws Exception {
        Pile pile = new Pile();

        totalAssertions++ ;
        assertEquals("new Pile() retourne une Pile", "Pile", pile.getClass()
().getName());
        bilanAssertions++ ;
    }

    /*
     Axiomes du type Pile
    */

    /*
     Préconditions du type Pile
    */

    /*
     main() de la classe de Test
    */
    public static void main(String[] args) {
        junit.textui.TestRunner.run(new TestSuite(PileTest.class));
        if (bilanAssertions == totalAssertions) { System.out.print("Bravo !"); }
        System.out.println(" "+bilanAssertions+"/"+totalAssertions+" assertions v
érfifiées");
    } // fin main
} // fin PileTest
```

2. Si vous utilisez SciTE au lieu de **eclipse**, pour compiler le programme de Test n'oubliez pas de

placer les fichiers **SciTE.properties** et **junit.jar** dans le répertoire de vos sources (avant d'ouvrir SciTE) ou bien exécutez ceci :

```
javac -cp .;junit.jar PileTest.java
```

3. Compléter la classe **Pile** pour qu'elle fournisse un enregistrement possédant :
 - a. un champ **elements** de type **String[]** et de taille 100
 - b. un champ **indiceSommet** de type **int**, initialisé à -1

9.4. Étape 2 : Mise en place des opérations du type abstrait **Pile**

1. Créer un fichier **PileOperations.java** et y placer une classe **PileOperations** contenant la fonction **empiler()** suivante :

```
public static Pile empiler(Pile pfPile, String pfElement) {  
    return(pfPile);  
}
```

2. Compiler **Pile.java** puis vérifier que le test passe toujours
3. Ajouter la fonction de test suivante à **PileTest.java**

Seconde fonction de test

```
public void test_type_empiler() throws Exception {  
    Pile pile = new Pile() ;  
  
    totalAssertions++ ;  
    assertEquals("empiler(pile,'XXX') retourne une Pile", "Pile", PileOperations  
.empiler(pile, "XXX").getClass().getName());  
    bilanAssertions++ ;  
}
```

Comprendre cette seconde fonction de test, compiler **PileTest.java** puis vérifier que les 2 tests passent.

4. Ajouter la fonction de test suivante à **PileTest.java**

Troisième fonction de test

```
public void test_type_depiler() throws Exception {  
    Pile pile = new Pile() ;  
    PileOperations.empiler(pile, "XXX") ;  
  
    totalAssertions++ ;  
    assertEquals("depiler(pile) retourne une Pile", "Pile", PileOperations.depiler  
(pile).getClass().getName());  
    bilanAssertions++ ;  
}
```

Comprendre cette troisième fonction de test et compléter **PileOperations.java** pour que le troisième test passe.

5. Ajouter la fonction de test suivante à **PileTest.java**

Quatrième fonction de test

```
public void test_type_sommet() throws Exception {  
    Pile pile = new Pile() ;  
    PileOperations.empiler(pile, "XXX") ;  
  
    totalAssertions++ ;  
    assertEquals("sommet(pile) retourne une String", "java.lang.String",  
    PileOperations.sommet(pile).getClass().getName());  
    bilanAssertions++ ;  
}
```

Comprendre cette quatrième fonction de test et compléter LE MOINS POSSIBLE **PileOperations.java** pour que le quatrième test passe.



à ce stade, les opérations du type Pile respectent seulement la définition des types requis et produits. Les opérations ne font pas encore ce qu'elles doivent faire.
Patience ...

9.5. Étape 3 : Implémentation des axiomes du type abstrait Pile (si vous avez encore du temps)

L'implémentation **java** du type Pile utilise un tableau de chaînes et une variable indice indiquant le sommet actuel de la pile.

1. Que doit contenir le tableau interne d'une Pile à la suite des actions suivantes :

```
Pile p = new Pile();
PileOperations.empiler(p, "A");
PileOperations.empiler(p, "B");
PileOperations.depiler(p);
PileOperations.depiler(p);
```

2. Pourquoi le code précédent est-il équivalent à celui-ci :

```
Pile p = new Pile();
PileOperations.depiler(PileOperations.depiler(PileOperations.empiler(PileOperations
    .empiler(p, "A"), "B")));
```

3. Ajouter le code suivant à **PileTest.java**

Test des axiomes du type abstrait Pile

```
public void XXXtest_axiome1() {
    Pile pile = new Pile();

    totalAssertions++;
    assertTrue("Une nouvelle pile est vide", PileOperations.estVide(pile));
    bilanAssertions++;
}

public void XXXtest_axiome2() throws Exception {
    Pile pile = new Pile();
    PileOperations.empiler(pile, "XXX");

    totalAssertions++;
    assertFalse("Après empiler() : pile n'est pas vide", PileOperations.estVide
(pile));
    bilanAssertions++;
}

public void XXXtest_axiome3() throws Exception {
    Pile pile = new Pile();
    PileOperations.empiler(pile, "XXX");
    PileOperations.depiler(pile);

    totalAssertions++;
    assertTrue("Après empiler(), depiler() : pile est vide", PileOperations.
estVide(pile));
    bilanAssertions++;
}

public void XXXtest_axiome4() throws Exception {
    Pile pile = new Pile();
    PileOperations.empiler(pile, "XXX");
```

```

        totalAssertions++ ;
        assertEquals("Apres empiler(pile,\\"XXX\\") : Sommet == \\"XXX\\\"", "XXX",
PileOperations.sommet(pile));
        bilanAssertions++ ;
    }

public void XXXtest_axiome5() throws Exception {
    Pile pile = new Pile() ;
    PileOperations.empiler(pile,"000") ;
    PileOperations.empiler(pile,"XXX") ;
    PileOperations.depiler(pile) ;

    totalAssertions++ ;
    assertEquals("Apres empiler(pile,\\"000\\"), empiler(pile,\\"XXX\\"), depiler(pile)
: Sommet == \\"000\\\"", "000", PileOperations.sommet(pile));
    bilanAssertions++ ;
}

public void XXXtest_axiomes3et5() throws Exception {
    Pile pile = new Pile() ;
    PileOperations.depiler(PileOperations.empiler(PileOperations.empiler(pile,
"000"),"XXX")) ;

    totalAssertions++ ;
    assertEquals("Apres empiler(pile,\\"000\\"), empiler(pile,\\"XXX\\"), depiler(pile)
: Sommet == \\"000\\\"", "000", PileOperations.sommet(pile));
    bilanAssertions++ ;

    PileOperations.depiler(pile) ;

    totalAssertions++ ;
    assertTrue("Apres depiler(pile) : pile est vide", PileOperations.estVide(
pile));
    bilanAssertions++ ;
}

```

4. POUR chaque fonction de test FAIRE

- Activer la fonction en supprimant 'XXX'
- Comprendre l'objectif
- Modifier la classe PileOperations pour que le test passe
- Compiler, tester
- Retourner en c) autant que nécessaire

Indications



1. **empiler()** incrémente le sommet de pile, puis insère le nouvel élément à l'indice sommet de pile
2. **depiler()** décrémente le sommet de pile
3. **sommet()** retourne l'élément situé en sommet de pile
4. Vous supposerez dans cet exercice que la pile n'est jamais pleine

10. Le patron *Observer*



10.1. Motivation

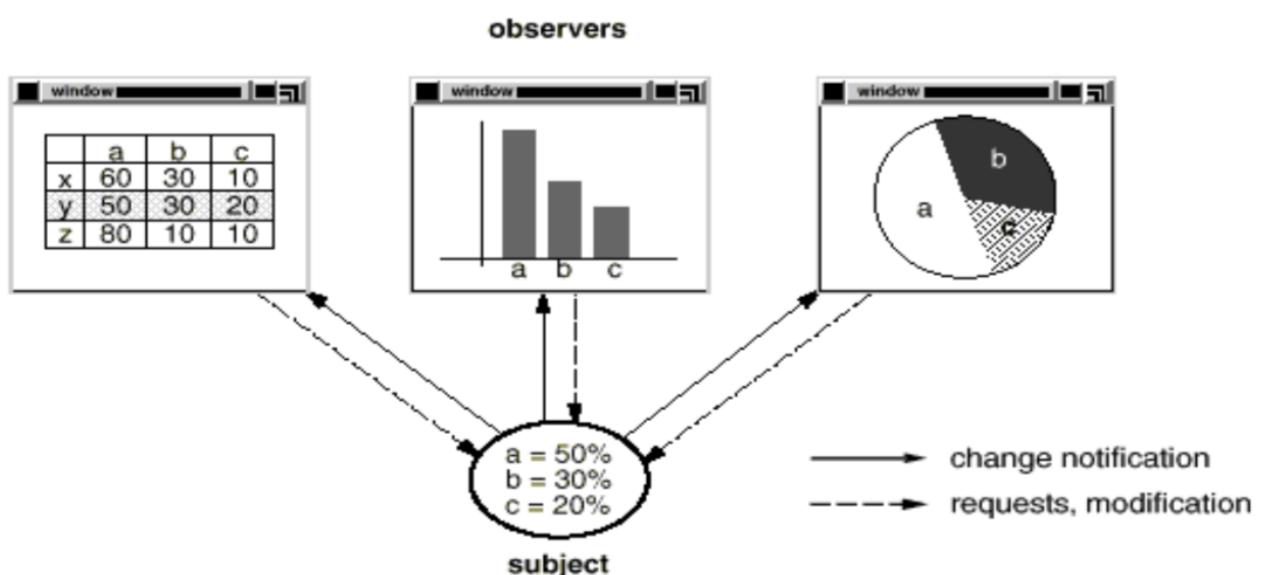


Figure 32. L'illustration classique du patron *Observer*

10.2. Définition

Observateur définit une relation entre objets de type un-à-plusieurs, de façon que, lorsqu'un objet change d'état, tous ceux qui en dépendent en soient notifiés et soient mis à jour automatiquement.

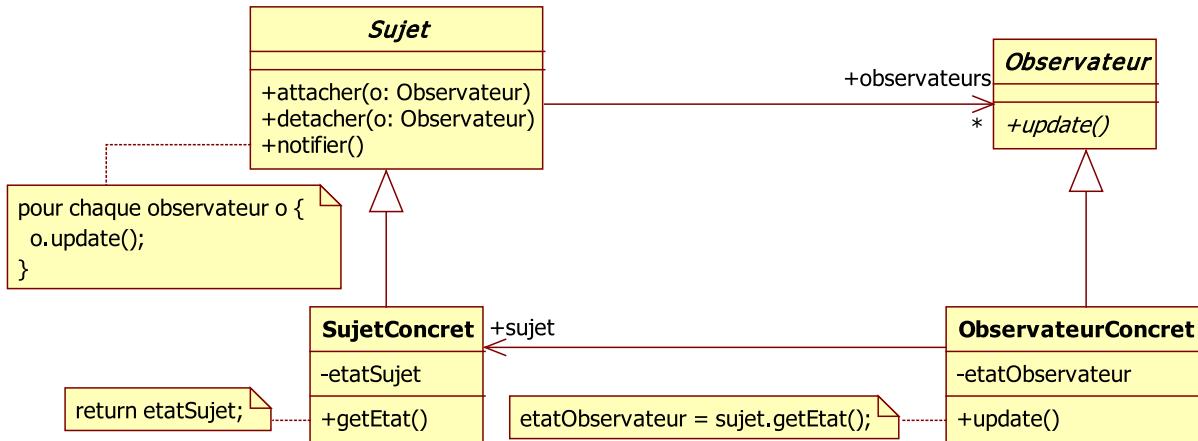


Figure 33. Modèle UML du patron Observer

10.3. Application

Le patron *Observer* est utilisable dans de nombreuses situations :

- Quand un concept a deux aspects, l'un dépendant de l'autre. Encapsuler ces aspects dans des objets séparés permet de les utiliser et les laisser évoluer de manière indépendante.
- Dès que le changement d'un objet entraîne le changement de plusieurs autres.
- Dès qu'un objet doit en notifier un certain nombre d'autres sans les connaître.

10.4. Observer en Java

Java fournit des classes *Observable/Observer* pour le patron *Observer*. La classe `java.util.Observable` est la classe de base pour les sujets. Ainsi, toute classe qui veut être observée étant cette classe voici les caractéristiques :

- fournit des méthodes pour ajouter/enlever des observateurs
- fournit des méthodes pour notifier les observateurs
- une sous-classe concrète doit seulement s'occuper de notifier à chaque méthode modifiant l'état des objets (*mutators*)
- utilise un vecteur stockant les références des observateurs

L'interface `java.util.Observer` correspond aux observateurs qui doivent implémenter cette interface.

10.4.1. La classe `java.util.Observable`

Voici la liste des méthodes de `java.util.Observable` :

```
public Observable()
public synchronized void addObserver(Observer o)
protected synchronized void setChanged()
public synchronized void deleteObserver(Observer o)
protected synchronized void clearChanged()
public synchronized boolean hasChanged()
public void notifyObservers(Object arg)
public void notifyObservers()
```

10.4.2. L'interface java.util.Observer

java.util.Observer

```
/**
 * This method is called whenever the observed object is changed. An
 * application calls an observable object's notifyObservers method to have all
 * the object's observers notified of the change.
 *
 * Parameters:
 * o - the observable object
 * arg - an argument passed to the notifyObservers method
 */
public abstract void update(Observable o, Object arg)
```

11. Une implémentation du MVC : les JTable java

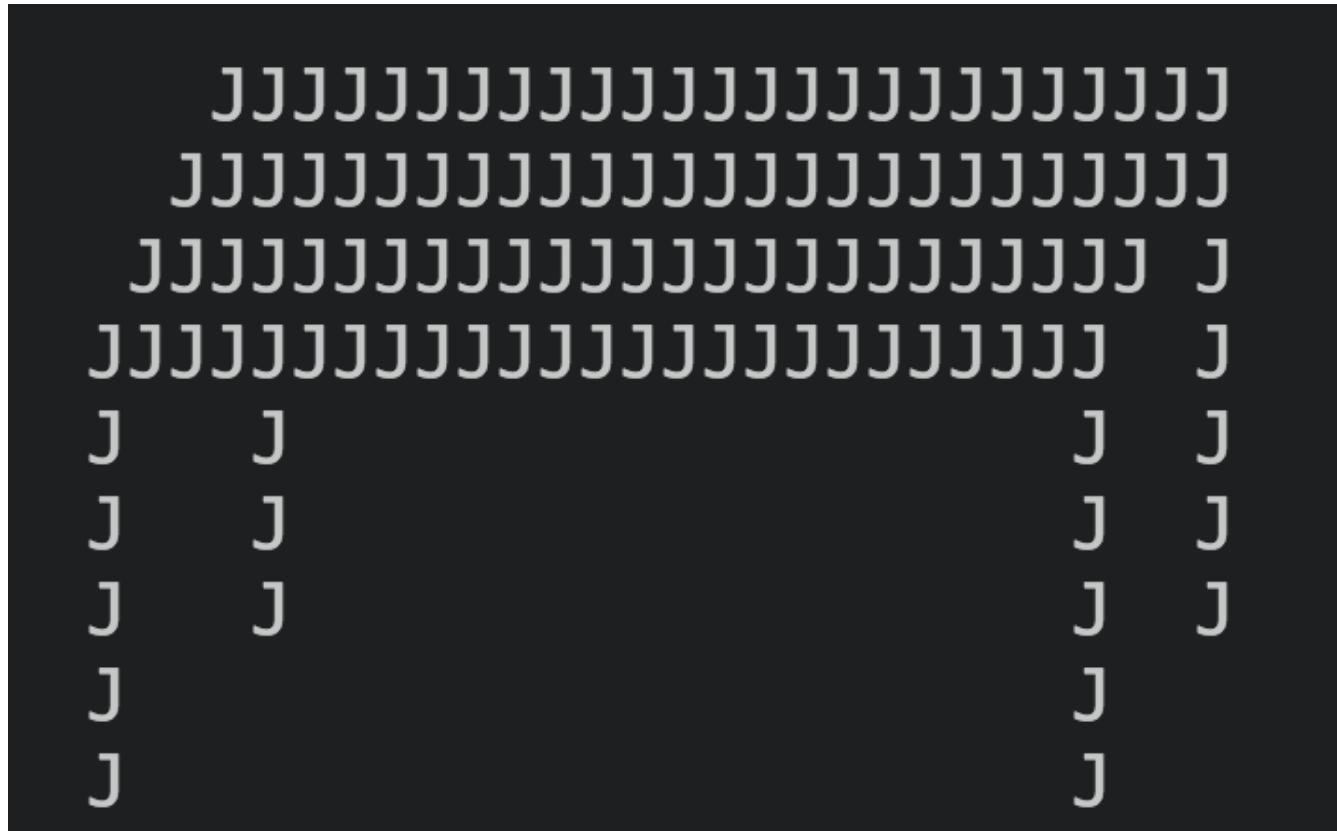
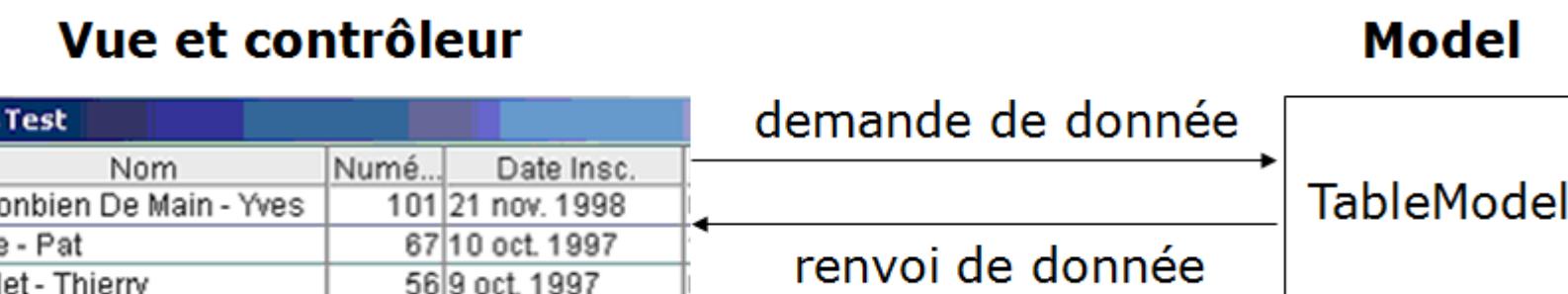
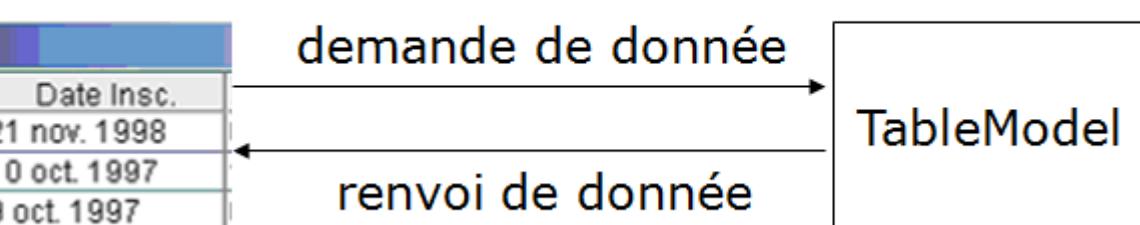


Figure 34. Ceci n'est pas une JTable

11.1. Le principe



ur



11.2. L'architecture

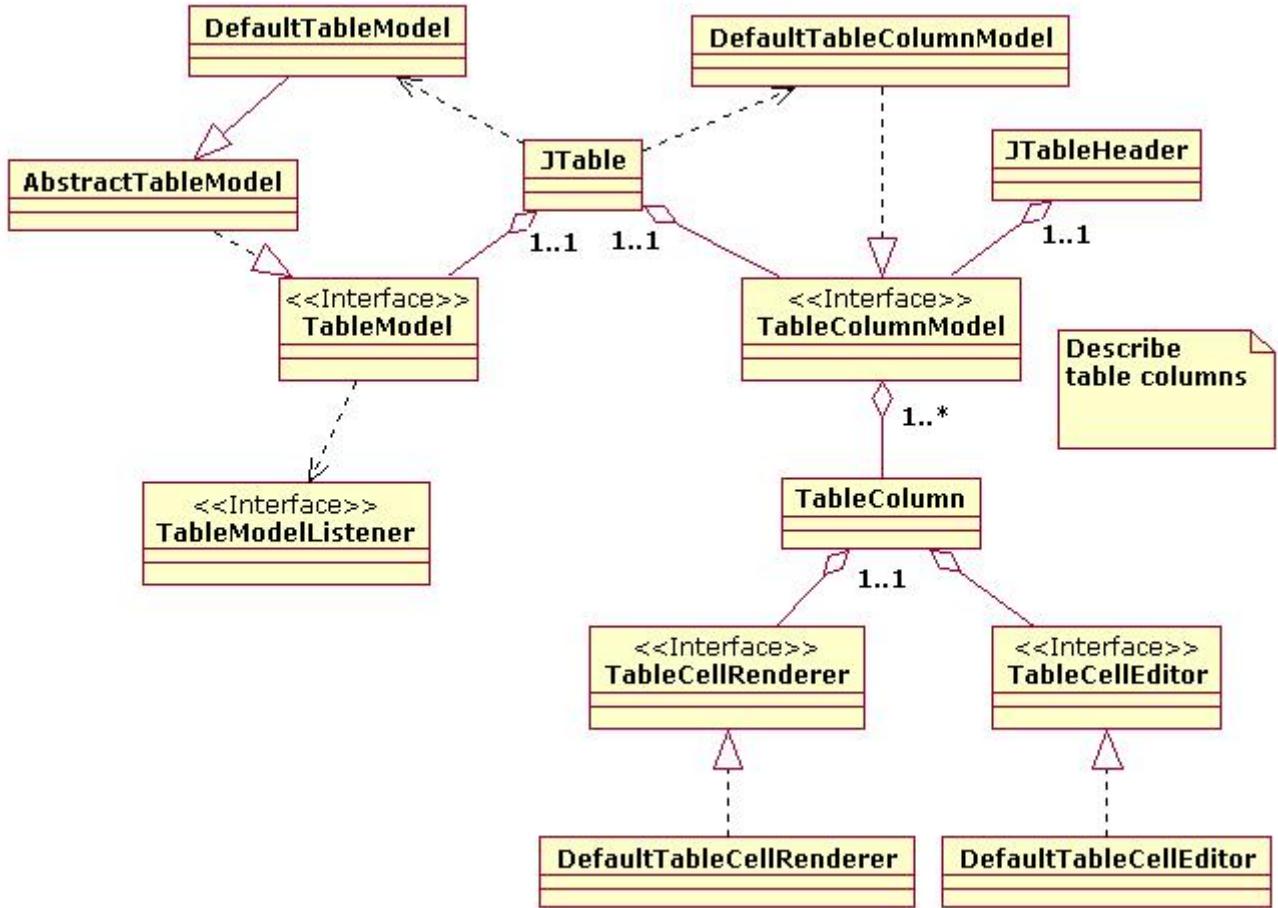


Figure 35. Architecture de `JTable`

12. Le patron Adaptateur



12.1. Le problème

On veut pouvoir :

- utiliser une classe existante, mais dont **l'interface ne coïncide pas** avec celle escomptée.
- créer une classe réutilisable qui collabore avec des classes sans relations avec elle et encore inconnues, c'est-à-dire avec des classes qui n'auront **pas nécessairement des interfaces compatibles**.
- vous avez besoin d'utiliser plusieurs sous-classes existantes, mais l'**adaptation de leur interface** par dérivation de chacune d'entre elles est impraticable. Un adaptateur objet peut adapter l'interface de sa classe parente.



Ce dernier cas ne concerne que le cas "adaptateur d'objet"

12.2. Exemple concret : le retour des canards

- L'existant :

```
public interface Canard {  
    public void cancaner();  
    public void voler();  
}  
  
public class Colvert implements Canard {  
    public void cancaner() {  
        System.out.println("Coincoin");  
    }  
    public void voler() {  
        System.out.println("Je vole");  
    }  
}
```

- Le "presque canard" :

```

public interface Dindon {
    public void glouglouter();
    public void voler();
}

public class DindonSauvage implements Dindon {
    public void glouglouter() {
        System.out.println("Glouglou");
    }
    public void voler() {
        System.out.println("Je ne vole pas loin");
    }
}

```

Vous êtes à court d'objets **Canard** et vous aimeriez utiliser des objets **Dindon** à la place!

```

public class AdaptateurDindon implements Canard {
    Dindon dindon;

    ...

    public void cancaner() {
        dindon.glouglouter();
    }

    public void voler() {
        // Adaptation du vol
        for(int i=0; i < 5; i++) {
            dindon.voler();
        }
    }
}

```

12.3. Le patron Adaptateur

Adaptateur (Adapter) permet de convertir l'interface d'une classe en une autre conformément à l'attente du client. L'Adaptateur permet à des classes de collaborer, alors qu'elles n'auraient pas pu le faire du fait d'interfaces incompatibles.

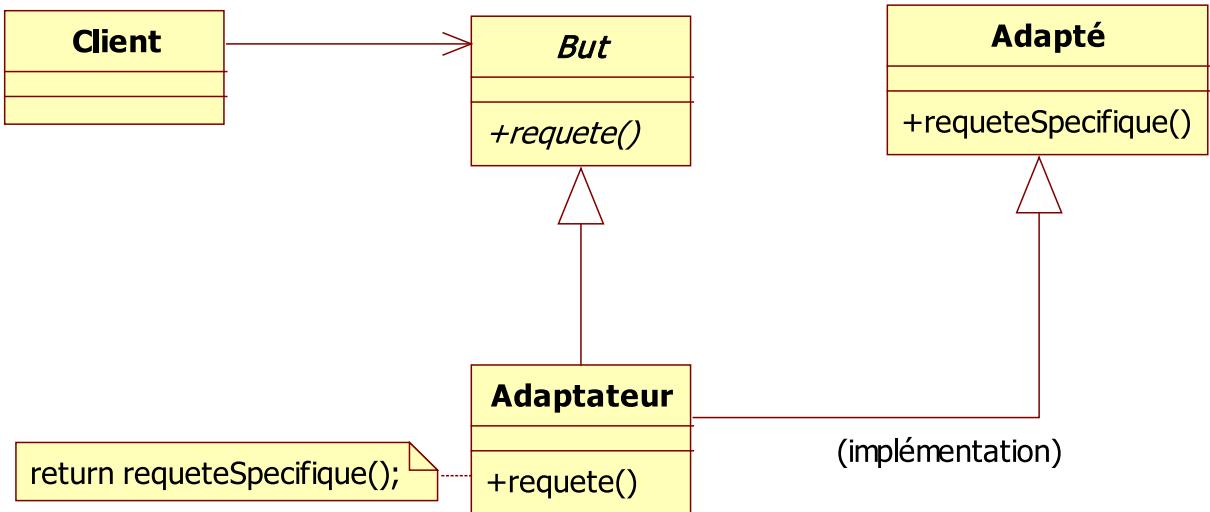


Figure 36. Modèle UML du patron Adaptateur

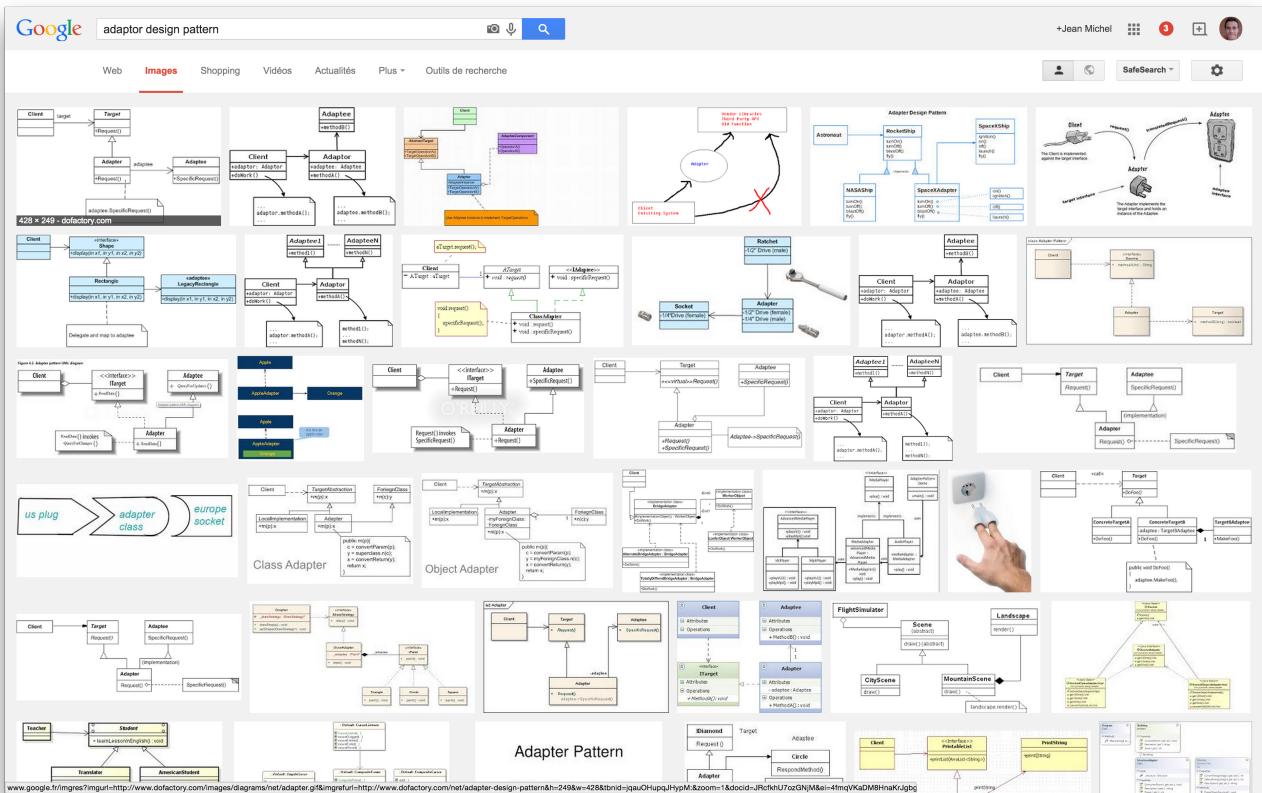


Figure 37. Adaptateur sur Google

13. Le patron Visiteur



13.1. Le problème

Quelques situations à problème :

- Une **structure** d'objets contient beaucoup de classes différentes d'interfaces distinctes, et vous désirez réaliser des opérations sur ces objets qui dépendent de leurs classes concrètes.
- Il s'agit d'effectuer plusieurs opérations distinctes et sans relation entre elles, sur les objets d'une structure, et ceci en **éitant de polluer leurs classes** avec ces opérations.
- Les classes qui définissent la structure objet changent rarement, mais on doit souvent **définir de nouvelles opérations** sur cette structure.

13.2. Illustration

Adapté d'un exemple tiré de http://www.tutorialspoint.com/design_pattern/visitor_pattern.htm

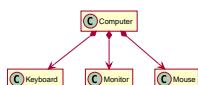


Figure 38. Une structure classique (sans patron)

13.2.1. Step 1

Définir une interface pour représenter les éléments de la structure.

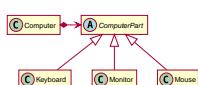


Figure 39. Implémentation d'une hiérarchie

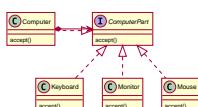


Figure 40. Utilisation d'une interface

ComputerPart.java (extrait)

```
public interface ComputerPart {  
    ...  
}
```

13.2.2. Step 2

Anticiper l'utilisation du visiteur.

ComputerPartVisitor.java (extrait)

```
public interface ComputerPartVisitor {  
    ...  
}
```

ComputerPart.java

```
public interface ComputerPart {  
    public void accept(ComputerPartVisitor computerPartVisitor);  
}
```

13.2.3. Step 3

Créer les classes concrètes qui implémentent l'interface.

Keyboard.java

```
public class Keyboard implements ComputerPart {  
  
    @Override  
    public void accept(ComputerPartVisitor computerPartVisitor) {  
        computerPartVisitor.visit(this);  
    }  
}
```

...

Computer.java

```
public class Computer implements ComputerPart {  
  
    ComputerPart[] parts;  
  
    public Computer(){  
        parts = new ComputerPart[] {new Mouse(), new Keyboard(), new Monitor()};  
    }  
  
    @Override  
    public void accept(ComputerPartVisitor computerPartVisitor) {  
        for (int i = 0; i < parts.length; i++) {  
            parts[i].accept(computerPartVisitor);  
        }  
        computerPartVisitor.visit(this);  
    }  
}
```

13.2.4. Step 4

Définir l'interface pour représenter le visiteur.

ComputerPartVisitor.java

```
public interface ComputerPartVisitor {  
    public void visit(Computer computer);  
    public void visit(Mouse mouse);  
    public void visit(Keyboard keyboard);  
    public void visit(Monitor monitor);  
}
```

13.2.5. Step 5

Créer des visiteurs concrets.

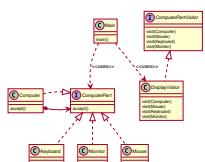


Figure 41. Visiteur : l'interface

```
public class DisplayVisitor implements ComputerPartVisitor {

    @Override
    public void visit(Computer computer) {
        System.out.println("Displaying Computer.");
    }

    @Override
    public void visit(Mouse mouse) {
        System.out.println("Displaying Mouse.");
    }

    @Override
    public void visit(Keyboard keyboard) {
        System.out.println("Displaying Keyboard.");
    }

    @Override
    public void visit(Monitor monitor) {
        System.out.println("Displaying Monitor.");
    }
}
```

13.2.6. Step 6

Utiliser le visiteur *DisplayVisitor*.

```
public class VisitorPatternDemo {
    public static void main(String[] args) {

        ComputerPart computer = new Computer();
        computer.accept(new DisplayVisitor());
    }
}
```

13.2.7. Step 7 (final)

Verify the output.

```
Displaying Mouse.
Displaying Keyboard.
Displaying Monitor.
Displaying Computer.
```

13.3. Le patron Visiteur

Visiteur (**Visitor**) permet la représentation d'une opération applicable aux éléments d'une structure d'objet.

Il définit une nouvelle opération, **sans qu'il soit nécessaire de modifier la classe** des éléments sur lesquels elle agit.

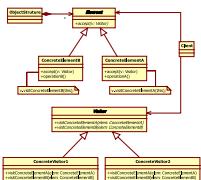


Figure 42. Patron Visiteur : structure



Figure 43. Patron Visiteur : comportement



Figure 44. Visiteur sur Google

13.4. Avantages/Inconvénients

Avantages :

- Permet d'ajouter des opérations à la structure d'un Composite **sans modifier la structure elle-même**.
- L'**ajout de nouvelles opérations** est relativement facile.
- Le code des opérations exécutées par le Visiteur est **centralisé**.

Inconvénients :

- L'encapsulation des classes du Composite est brisée.
- Comme une fonction de navigation est impliquée, les modifications de la structure du Composite sont plus difficiles.

13.5. Exemples d'utilisation

- calcul sur un ensemble structuré d'éléments
- génération de rapports ou de code
- ...

13.6. Exemple concret d'utilisation en Java



Exemple tiré de [ce site](#).

ItemElement.java

```
public interface ItemElement {  
  
    public int accept(ShoppingCartVisitor visitor);  
}
```

Book.java

```
public class Book implements ItemElement {  
  
    private int price;  
    private String isbnNumber;  
  
    public Book(int cost, String isbn){  
        this.price=cost;  
        this.isbnNumber=isbn;  
    }  
  
    public int getPrice() {  
        return price;  
    }  
  
    public String getIsbnNumber() {  
        return isbnNumber;  
    }  
  
    @Override  
    public int accept(ShoppingCartVisitor visitor) {  
        return visitor.visit(this);  
    }  
}
```

Fruit.java

```
public class Fruit implements ItemElement {

    private int pricePerKg;
    private int weight;
    private String name;

    public Fruit(int priceKg, int wt, String nm){
        this.pricePerKg=priceKg;
        this.weight=wt;
        this.name = nm;
    }

    public int getPricePerKg() {
        return pricePerKg;
    }

    public int getWeight() {
        return weight;
    }

    public String getName(){
        return this.name;
    }

    @Override
    public int accept(ShoppingCartVisitor visitor) {
        return visitor.visit(this);
    }
}
```

ShoppingCartVisitor.java

```
public interface ShoppingCartVisitor {

    int visit(Book book);
    int visit(Fruit fruit);
}
```

ShoppingCartVisitorImpl.java

```
public class ShoppingCartVisitorImpl implements ShoppingCartVisitor {  
  
    @Override  
    public int visit(Book book) {  
        int cost=0;  
        //apply 5$ discount if book price is greater than 50  
        if(book.getPrice() > 50){  
            cost = book.getPrice()-5;  
        } else cost = book.getPrice();  
        System.out.println("Book ISBN::"+book.getIsbnNumber() + " cost =" +cost);  
        return cost;  
    }  
  
    @Override  
    public int visit(Fruit fruit) {  
        int cost = fruit.getPricePerKg()*fruit.getWeight();  
        System.out.println(fruit.getName() + " cost = " +cost);  
        return cost;  
    }  
}
```

ShoppingCartClient.java

```
public class ShoppingCartClient {  
  
    public static void main(String[] args) {  
        ItemElement[] items = new ItemElement[]{new Book(20, "1234"), new Book(100,  
"5678"),  
        new Fruit(10, 2, "Banana"), new Fruit(5, 5, "Apple")};  
  
        int total = calculatePrice(items);  
        System.out.println("Total Cost = "+total);  
    }  
  
    private static int calculatePrice(ItemElement[] items) {  
        ShoppingCartVisitor visitor = new ShoppingCartVisitorImpl();  
        int sum=0;  
        for(ItemElement item : items){  
            sum = sum + item.accept(visitor);  
        }  
        return sum;  
    }  
}
```

```

Book ISBN::1234 cost =20
Book ISBN::5678 cost =95
Banana cost = 20
Apple cost = 25
Total Cost = 160

```

14. Le patron proxy

14.1. Le problème

On a besoin de références à un objet, qui soient plus **créatives** et plus **sophistiquées** qu'un simple pointeur.

14.2. Le patron Proxy

Procuration (Proxy) fournit à un tiers un mandataire ou un remplaçant, pour contrôler l'accès à cet objet.



Figure 45. Modèle UML du patron Proxy

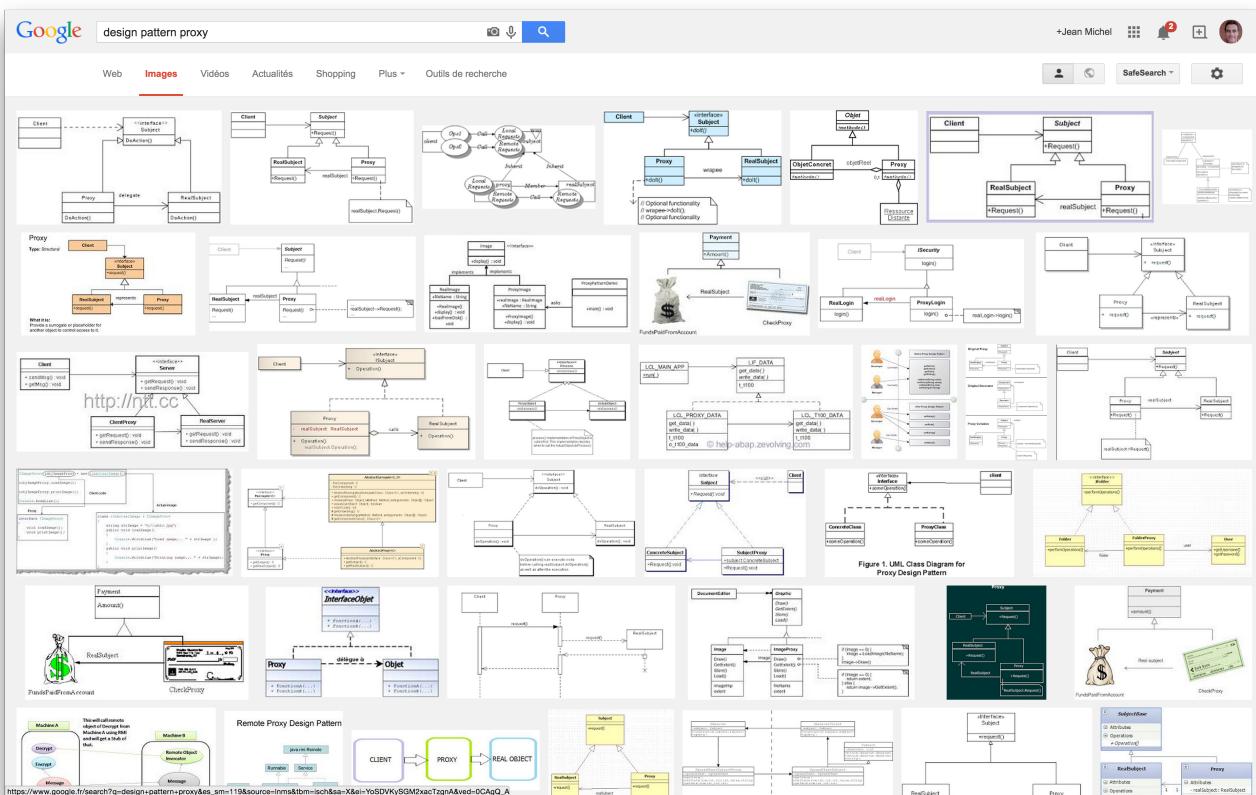


Figure 46. Proxy sur Google

14.3. Utilisations

- Une *procuration à distance* fournit un représentant local d'un objet situé dans un espace adresse différent.
- Une *procuration virtuelle* crée des objets lourds à la demande.
- Une *procuration de protection* contrôle l'accès à l'objet original. Les procurations de protection sont utiles quand les objets doivent satisfaire différents droits d'accès.
- Une *référence intelligente* est le remplaçant d'un pointeur brut, qui réalise des opérations supplémentaires, lors de l'accès à l'objet. Quelques utilisations typiques sont :
 - décompte du nombre des références faites à un objet réel, de sorte que celui-ci puisse être libéré automatiquement, dès qu'il n'y a plus de références ;
 - charger en mémoire un objet persistant quand il est référencé pour la première fois ;
 - vérifier, avant d'y accéder, que l'objet réel est verrouillé, pour être sûr qu'aucun autre objet ne pourra le changer.

14.4. Exemple concret : RMI

Remote Method Invocation

```
import java.rmi.*;  
public interface MonService extends Remote {  
    public String direBonjour() throws RemoteException;  
}
```

```
import java.rmi.*;  
import java.rmi.server.*;  
  
public class MonServiceImpl extends UnicastRemoteObject implements MonService {  
    public String direBonjour() {  
        return "Le serveur dit 'Bonjour'";  
    }  
    public MonServiceImpl() throws RemoteException {}  
    public static void main (String[] args) {  
        try {  
            MonService service = new MonServiceImpl();  
            Naming.rebind("BonjourDistant", service);  
        } catch(Exception ex) {  
            ex.printStackTrace();  
        }  
    }  
}
```

```

MonService service =
    (MonService) Naming.lookup("rmi://127.0.0.1/BonjourDistant");
...
service.direBonjour();

```

15. Le patron Itérateur

15.1. Le problème

On veut pouvoir :

- pour accéder au contenu d'un objet d'un agrégat sans en révéler la représentation interne ;
- pour gérer simultanément plusieurs parcours dans des agrégats d'objets ;
- pour offrir une interface uniforme pour les parcours au travers de diverses structures agrégats (c'est-à-dire, pour permettre l'itération polymorphe).

15.2. Le patron Itérateur

Itérateur (Iterator) fournit un moyen d'accès séquentiel aux éléments d'un agrégat d'objets, sans mettre à découvert la représentation interne de celui-ci.

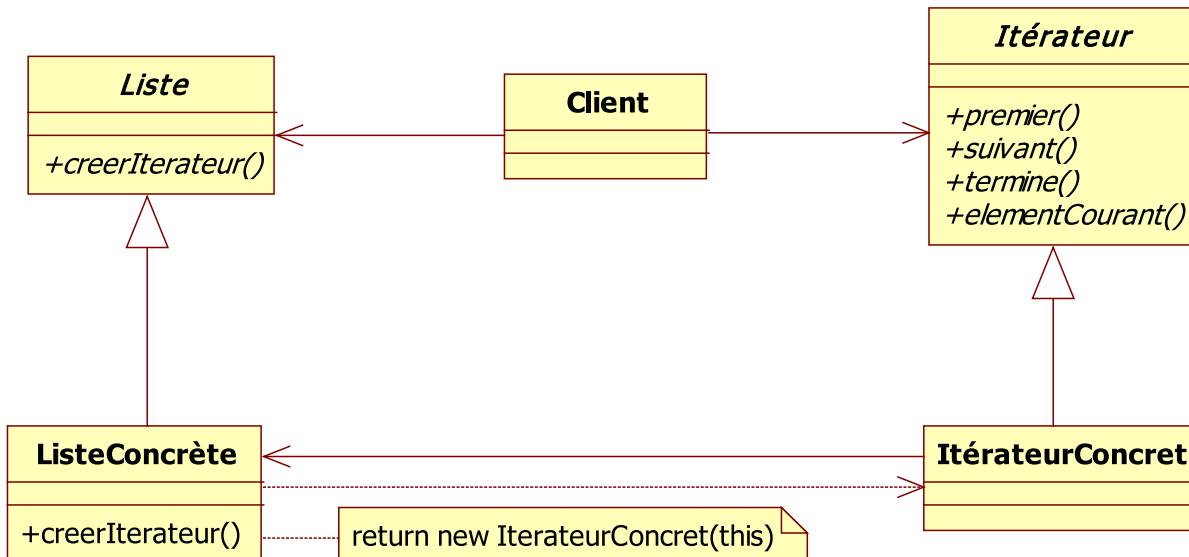


Figure 47. Modèle UML du patron Itérateur

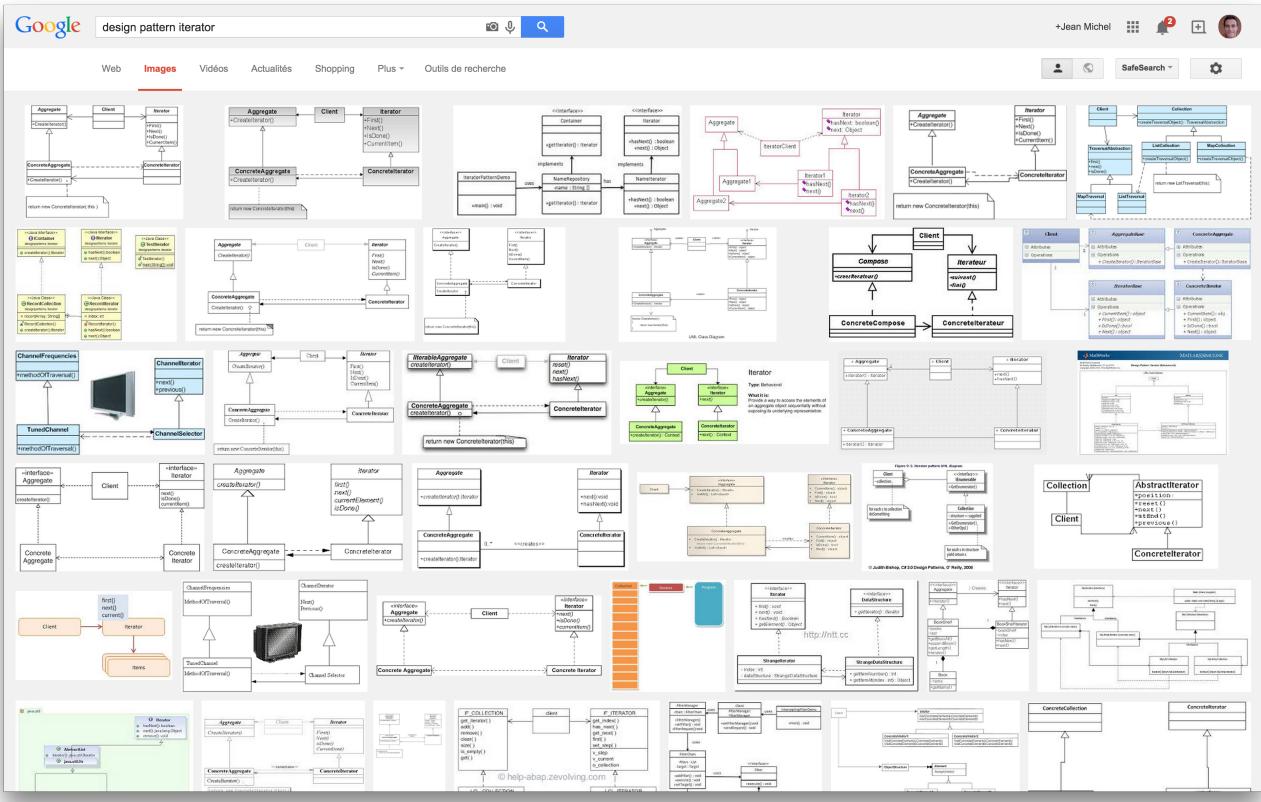


Figure 48. Iterateur sur Google

15.3. Exemple concret

```
# Saluer tout le monde
def say_hi
  if @names.nil?
    puts "..."
  elsif @names.respond_to?("each")
    # @names est une liste de noms : traitons-les uns par uns
    @names.each do |name|
      puts "Hello #{name}!"
    end
  else
    puts "Hello #{@names}!"
  end
end
```

16. Le patron Composite

16.1. Le problème

On veut pouvoir :

- représenter des hiérarchies de l'individu.

- que le client n'ait pas à se préoccuper de la différence entre "combinaisons d'objets" et "objets individuels". Les clients pourront traiter de façon uniforme tous les objets de la structure composite.

16.2. Le patron Composite

Composite permet de composer des objets en des structures arborescentes pour représenter des hiérarchies composant/composé. Permet au client de traiter d'une façon unique les objets et les combinaisons d'objets.

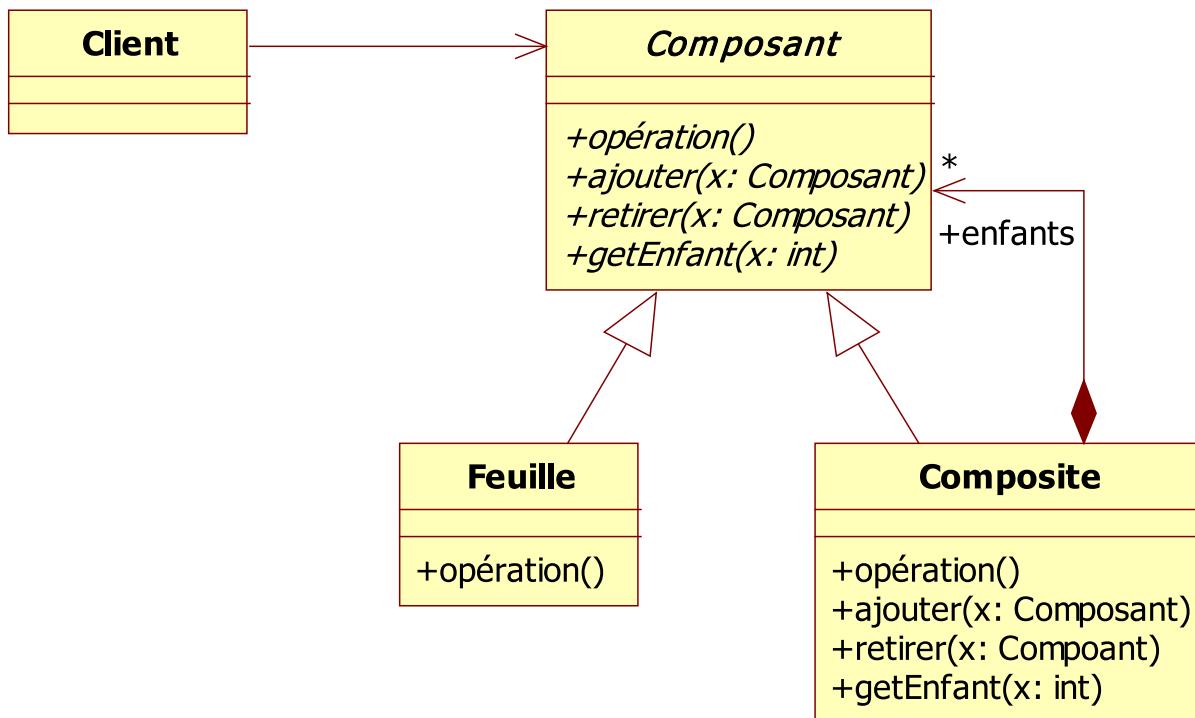


Figure 49. Modèle UML du patron Composite

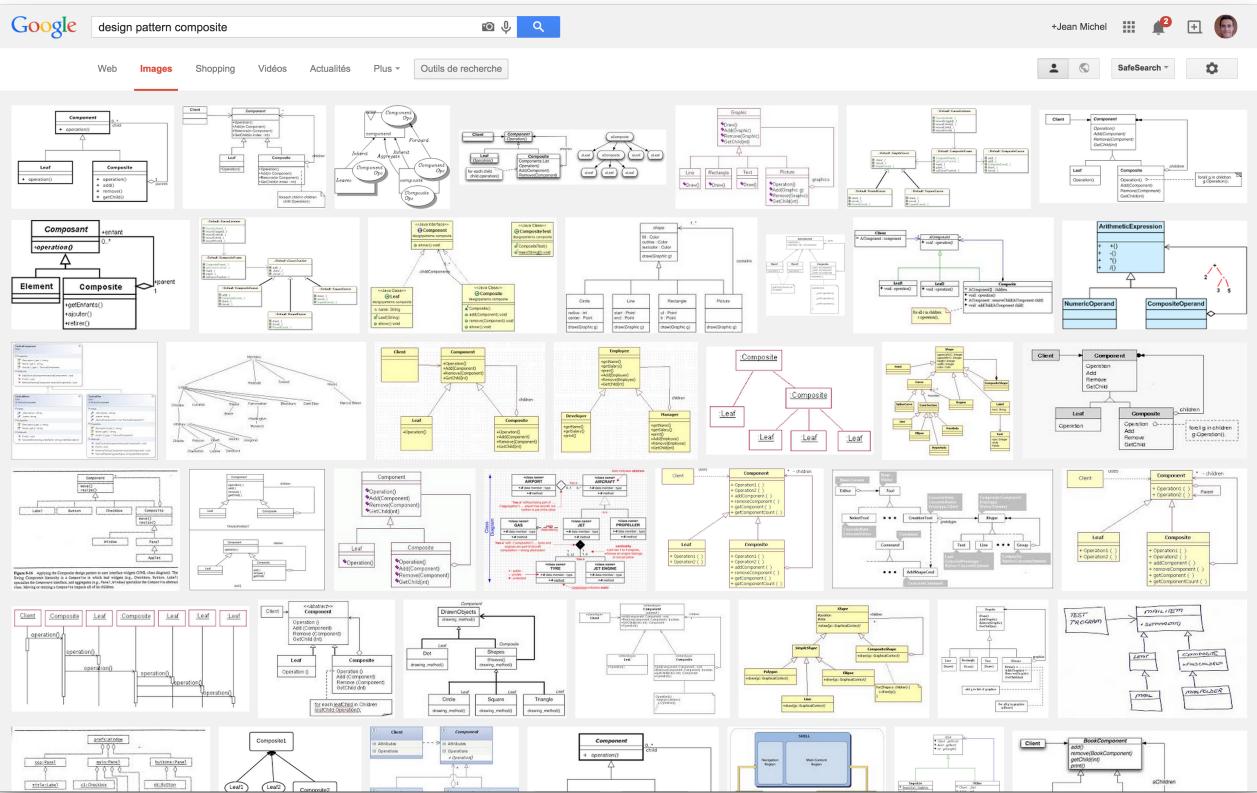


Figure 50. Composite sur Google

16.3. Exemple concret

```
import java.util.ArrayList;

interface Graphic {
    public void print();
}

class CompositeGraphic implements Graphic {

    private ArrayList<Graphic> mChildGraphics = new ArrayList<Graphic>();

    public void print() {
        for (Graphic graphic : mChildGraphics) {
            graphic.print();
        }
    }

    public void add(Graphic graphic) {
        mChildGraphics.add(graphic);
    }

    public void remove(Graphic graphic) {
        mChildGraphics.remove(graphic);
    }
}
```

16.4. un "Anti" exemple

Que pensez-vous de cette définition de Composite ?

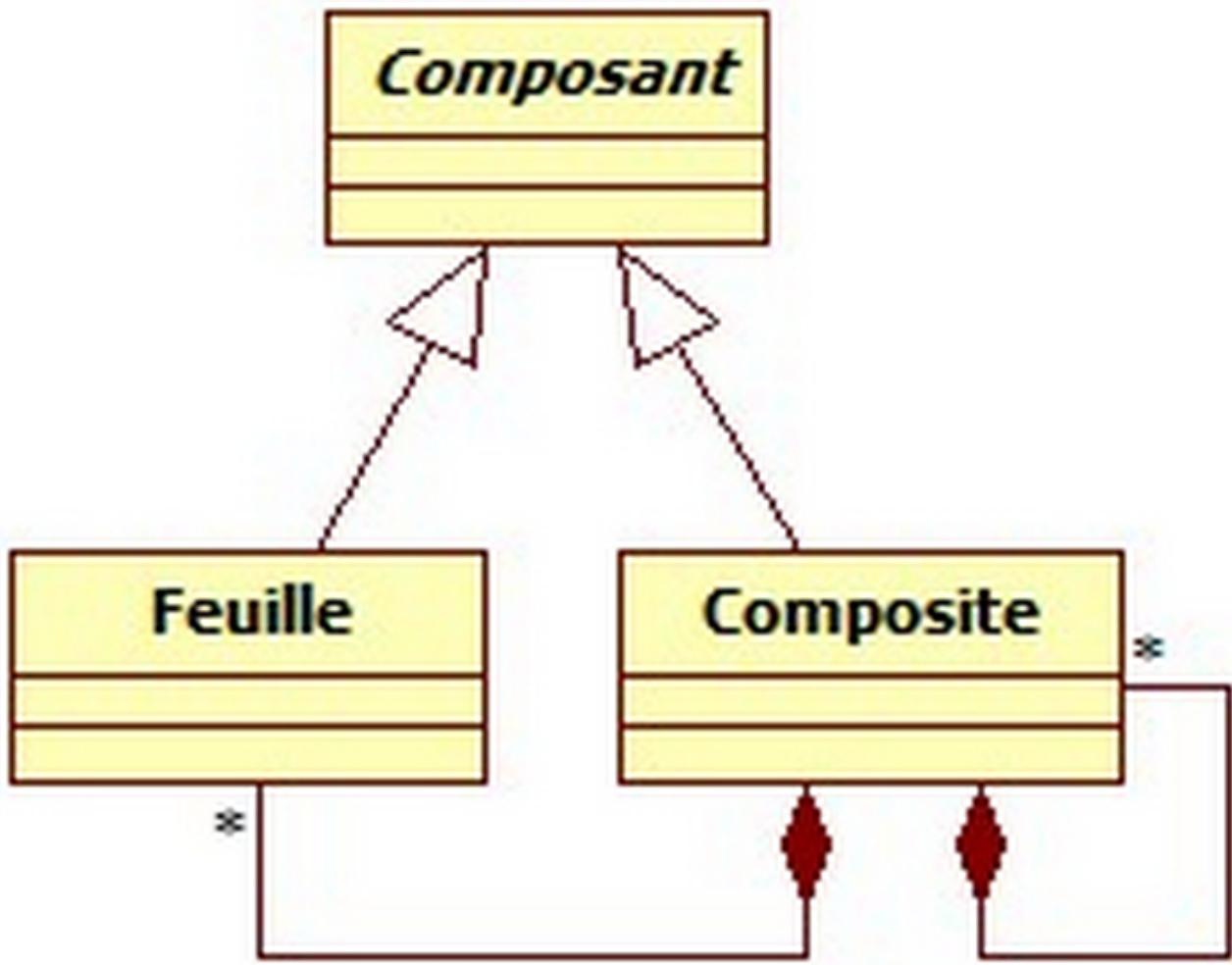


Figure 51. Patron abîmé composite



On appelle ces modèles des "Patrons abîmés" (*anti-patterns*).

17. Retour sur le refactoring Banque

17.1. Le problème

```

continuer = true;
while (continuer) {
    ApplicationAgenceBancaire.afficherMenu(monAg);
    choix = lect.next();
    choix = choix.toLowerCase();
    switch (choix) {
        case "q" :
            System.out.println("ByeBye");
            ApplicationAgenceBancaire.tempo();
            continuer = false;
            break;
        case "l" :
            monAg.afficher();
            ApplicationAgenceBancaire.tempo();
            break;
        case "v" :
            System.out.print("Num compte -> ");
            numero = lect.nextInt();
            c = monAg.getCompte(numero);
            if (c==null) {
                System.out.println("Compte inexistant ...");
            } else {
                c.afficher();
            }
            ApplicationAgenceBancaire.tempo();
            break;
        case "p" :
            System.out.print("Propriétaire -> ");
            nom = lect.next();
            ApplicationAgenceBancaire.comptesDUnPropretaire (monAg, nom);
            ApplicationAgenceBancaire.tempo();
            break;
        case "d" :
            System.out.print("Num compte -> ");
            numero = lect.nextInt();
            System.out.print("Montant à déposer -> ");
            montant = lect.nextDouble();
            ApplicationAgenceBancaire.deposerSurUnCompte(monAg, numero, montant);
            ApplicationAgenceBancaire.tempo();
            break;
        case "r" :
            System.out.print("Num compte -> ");
            numero = lect.nextInt();
            System.out.print("Montant à retirer -> ");
            montant = lect.nextDouble();
            ApplicationAgenceBancaire.retirerSurUnCompte(monAg, numero, montant);
            ApplicationAgenceBancaire.tempo();
            break;
        default :
            System.out.println("Erreur de saisie ...");
            ApplicationAgenceBancaire.tempo();
            break;
    }
}

```



Remplacer tous ces `switch cases`

```

continuer = true;
while (continuer) {
    AAB.afficherMenu(monAg);
    choix = lect.next();
    choix = choix.toLowerCase();
    switch (choix) {
        ...
        case "p" :
            System.out.print("Propriétaire -> ");
            nom = lect.next();
            AAB.comptesDUnPropretaire (monAg, nom);
            break;
        ...
    }
}

```

- Afficher une liste séparemment du switch

```

//AAB.afficherMenu(monAg);
System.out.println("Menu de " + ag.getNomAgence() + " (" + ag.getLocAgence() +
");
System.out.println("l - Liste des comptes de l'agence");
...
System.out.println("p - voir les comptes d'un Propriétaire (par son nom)");
...
System.out.print("Choix -> ");
}

```

- Tester tous les choix pour actionner la bonne option

```

...
case "p" :
    System.out.print("Propriétaire -> ");
    nom = lect.next();
    AAB.comptesDUnPropretaire (monAg, nom);
    break;
...

```

17.2. Une solution

- Des listes
- Des options de menu qui encapsulent l'action à réaliser

```
public interface ActionList extends Action {  
  
    public String listTitle();  
    public int size();  
  
    public boolean addAction(Action ac);  
    public boolean removeAction(Action ac);  
  
    public String[] listOfActions() ;  
  
}
```

Une interface pour les options de menu

```
public interface Action {  
  
    public String actionMessage ();  
    public void execute(AgenceBancaire ag);  
}
```

Une classe concrète par option de menu

```
public class Action1 implements Action {  
  
    private String lineMessage;  
  
    ...  
    public String actionMessage() {  
        return this.lineMessage;  
    }  
  
    public void execute(AgenceBancaire ab) {  
  
        ...  
        ab.afficher();  
    }  
}
```

Utilisation de l'action

```
action.execute(ab);
```

Lien entre liste et action

```
Action a1 = new Action1("Liste des comptes de l'agence");
Action a2 = new Action2("Voir un compte (par son numéro)");
Action a3 = new Action3(...);

ActionList al1 = new ActionListAgenceBancaire("Menu Général");

al1.addAction(a1);
al1.addAction(a2);
```

Lien entre liste et action : choix dans la liste

```
public void execute(AgenceBancaire ab) throws Exception {
    ...
    while (true) {
        this.printMenu();

        choice = this.readResponse();
        ...
        this.myMenu.get(choice).execute(ab);
        ...
    }
}
```

La liste peut elle-même être une option de menu (une action)!

```
public interface ActionList extends Action {
}
```

18. Pour aller plus loin avec les patrons...



18.1. Partagez votre vocabulaire

- Dans les réunions de conception (pas nécessairement avec le client)
- Avec les autres développeurs
- Dans la documentation de votre architecture
- Dans les commentaires du code et les conventions de nommage
- Dans les groupes/blogs de développeurs
- (pas pendant les exams!)

18.2. Ne foncez pas tête baissée

Quelques conseils :

- Les patterns sont des **outils, non des règles.**
⇒ Rien n'empêche de les modifier et de les adapter à votre problème.
- Ne visez l'extensibilité **que si la question se pose réellement** dans la pratique, pas si elle est uniquement hypothétique.
- Ne vous emballez pas et **recherchez la simplicité.**
⇒ Si vous trouvez une solution plus simple que l'emploi d'un pattern, n'hésitez pas !
- éliminez ce qui n'est pas vraiment nécessaire.
⇒ N'ayez pas peur de **supprimer un design pattern inutile** de votre conception.

18.3. Les autres types de patrons

Il n'y a pas que les 3 types de patrons que l'on a vu :

- De création
- Structurels
- Comportementaux

Il y a par exemple :

- Les patrons d'architecture
- Les patrons d'application
- Les patrons de domaine
- Les patrons de processus
- Les patrons d'organisation
- Les patrons de conception d'interfaces utilisateur

18.4. Les anti-patrons



- Des solutions souvent **appliquées à tort** à des problèmes récurrents
- Décrit comment partir d'un problème pour arriver à une **mauvaise solution**

- Vous dit **pourquoi** une mauvaise solution est attrayante
- Suggère d'autres patrons applicables pouvant fournir de meilleures solutions

18.5. Tous les patrons qu'on a pas vu

Il y en a beaucoup :

- Chaîne de responsabilité
- Commande
- Décorateur
- Façade
- Interprète
- Médiateur
- Memento
- Monteur
- Patron de méthode
- Poids-mouche
- Pont
- Prototype

18.6. Principale utilisation des patrons : refactoring!

⇒ Les 2 prochaines semaines vous allez refactorer une application :

- en binôme
- en testant
- en documentant
- en justifiant
- en essayant pas à tout prix de caser le plus de patrons "au kilo"

Glossaire et définition



Ces définitions seront enrichies au fur et à mesure des patrons étudiés.

Patrons de création

Singleton

Singleton garantit qu'une classe n'a qu'une seule instance et fournit un point d'accès global à cette instance.

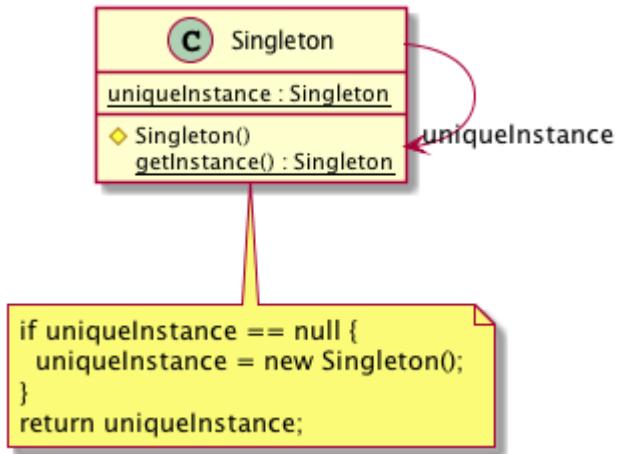


Figure 52. Modèle UML du patron Singleton

Fabrique

Fabrique (simple) définit une interface pour la création d'un objet, mais en laissant à des sous-classes le choix des classes à instancier (voir aussi [Fabrique abstraite](#)).

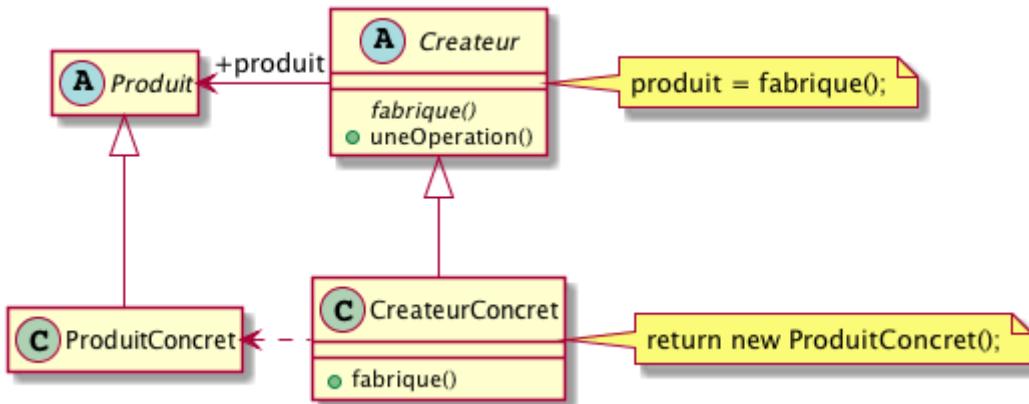


Figure 53. Modèle UML du patron Fabrique

Fabrique abstraite

Fabrique (abstraite) fournit une interface pour la création de familles d'objets apparentés ou interdépendants, sans qu'il soit nécessaire de spécifier leurs classes concrètes (voir aussi [Fabrique](#)).

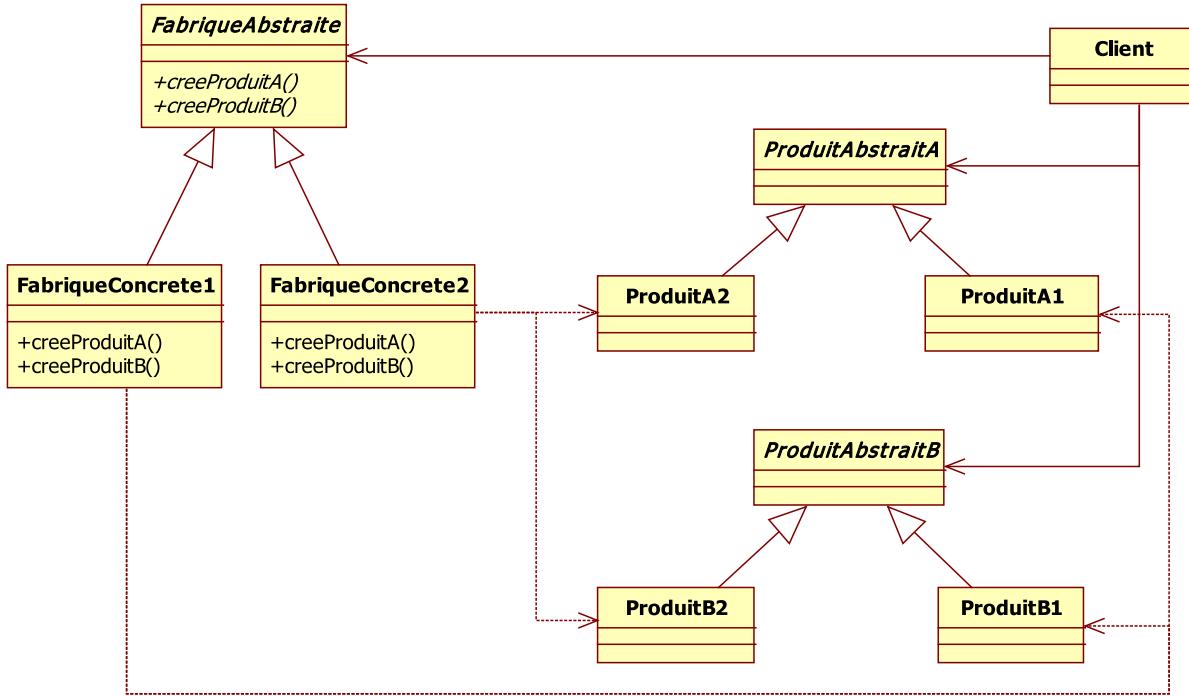


Figure 54. Modèle UML du patron Fabrique Abstraite

Patrons comportementaux

État ($State^{uk}$)

Etat permet à un objet de modifier son comportement, quand son état interne change. Tout se passe comme si l'objet changeait de classe.

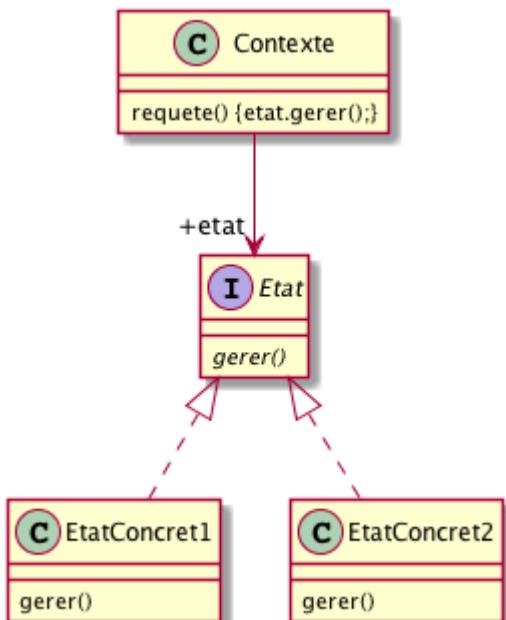


Figure 55. Modèle UML du patron Etat

Itérateur ($Iterator^{uk}$)

Itérateur (**Iterator**) fournit un moyen d'accès séquentiel aux éléments d'un agrégat d'objets, sans mettre à découvert la représentation interne de celui-ci.

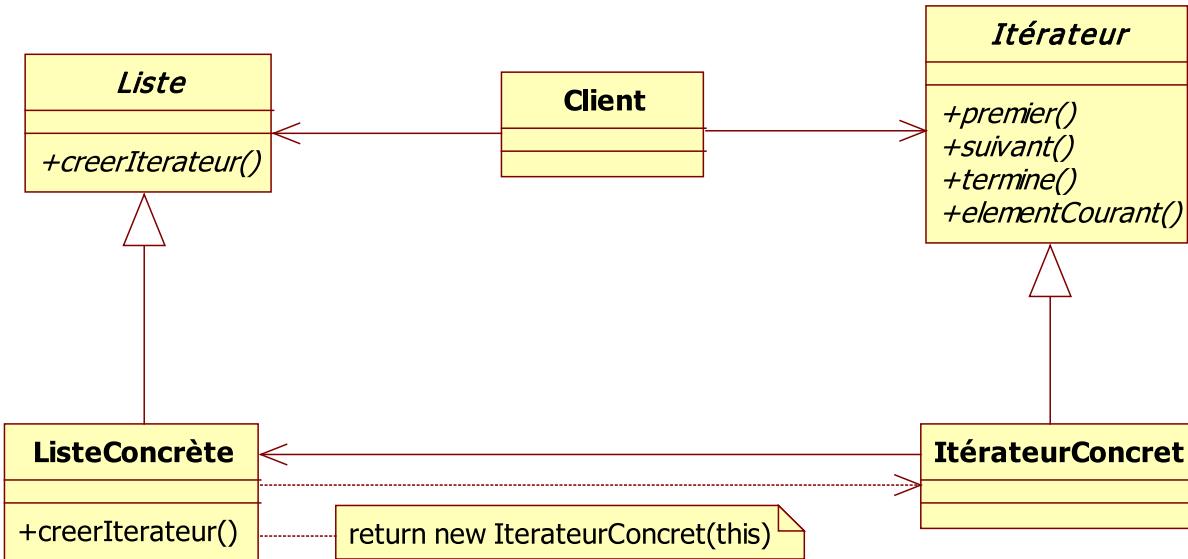


Figure 56. Modèle UML du patron Itérateur

Observateur (Observer^{uk})

Observateur définit une relation entre objets de type un-à-plusieurs, de façon que, lorsqu'un objet change d'état, tous ceux qui en dépendent en soient notifiés et soient mis à jour automatiquement.

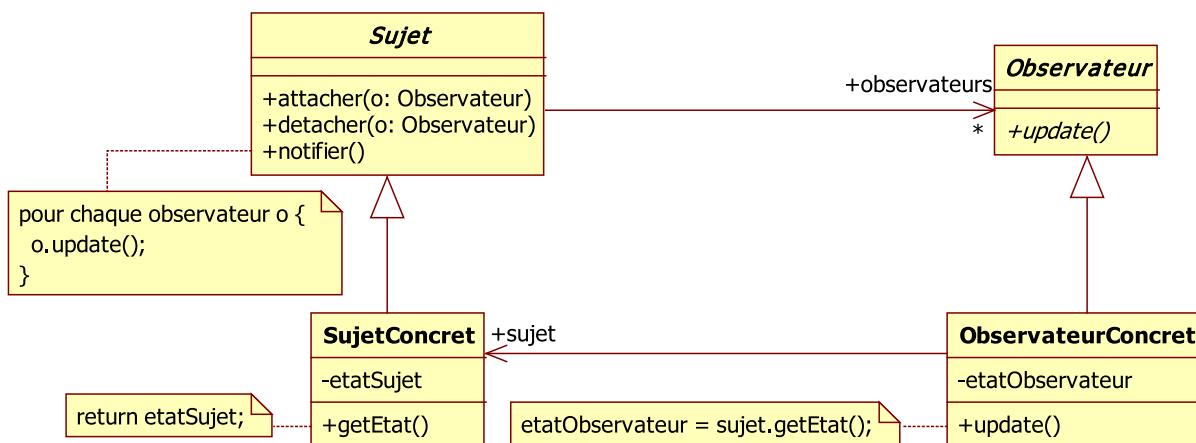


Figure 57. Modèle UML du patron Observer

Stratégie (Strategy^{uk})

Stratégie définit une famille d'algorithmes, encapsule chacun d'eux et les rend interchangeables. Il permet à l'algorithme de varier indépendamment des clients qui l'utilisent.

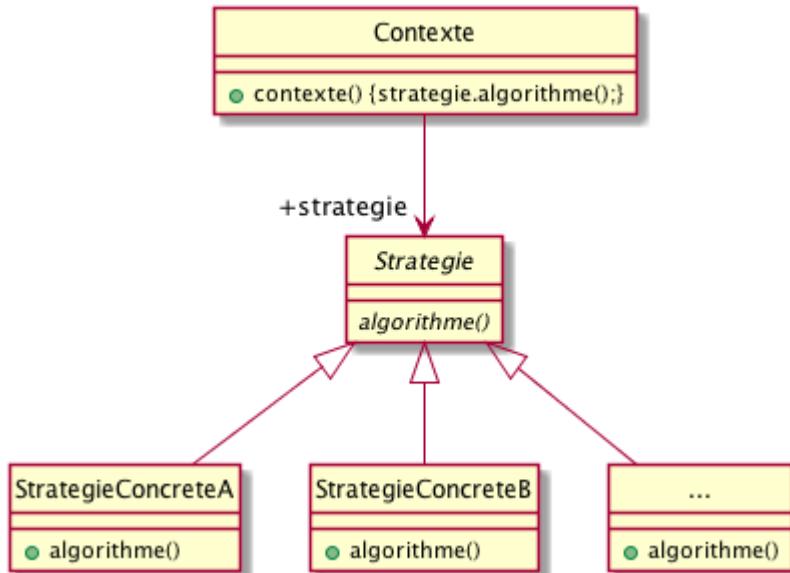


Figure 58. Modèle UML du patron Strategy

Visiteur (*Visitor*^{uk})

Visiteur (Visitor) permet la représentation d'une opération applicable aux éléments d'une structure d'objet. Il définit une nouvelle opération, sans qu'il soit nécessaire de modifier la classe des éléments sur lesquels elle agit.

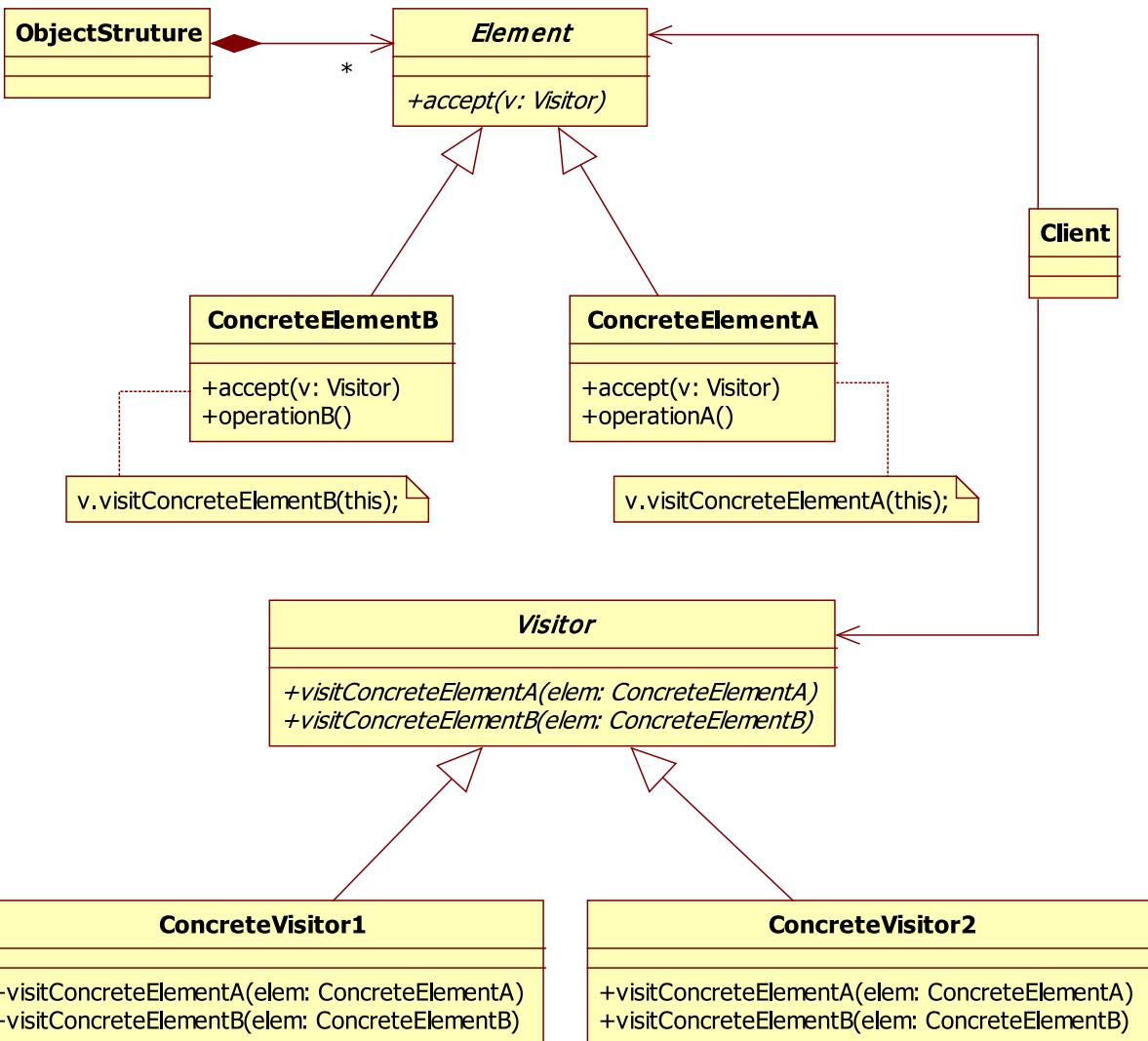


Figure 59. Patron Visiteur : structure

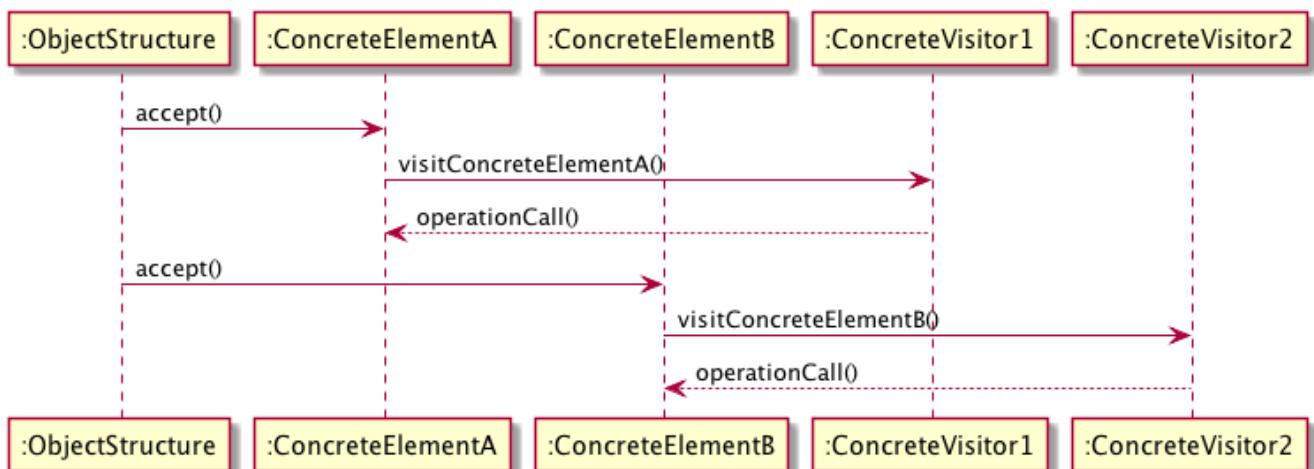


Figure 60. Patron Visiteur : comportement

Patrons structurels

Adaptateur (Adaptor^{uk})

Adaptateur (Adaptor) permet de convertir l'interface d'une classe en une autre conformément à l'attente du client. L'Adaptateur permet à des classes de collaborer, alors qu'elles n'auraient

pas pu le faire du fait d'interfaces incompatibles.

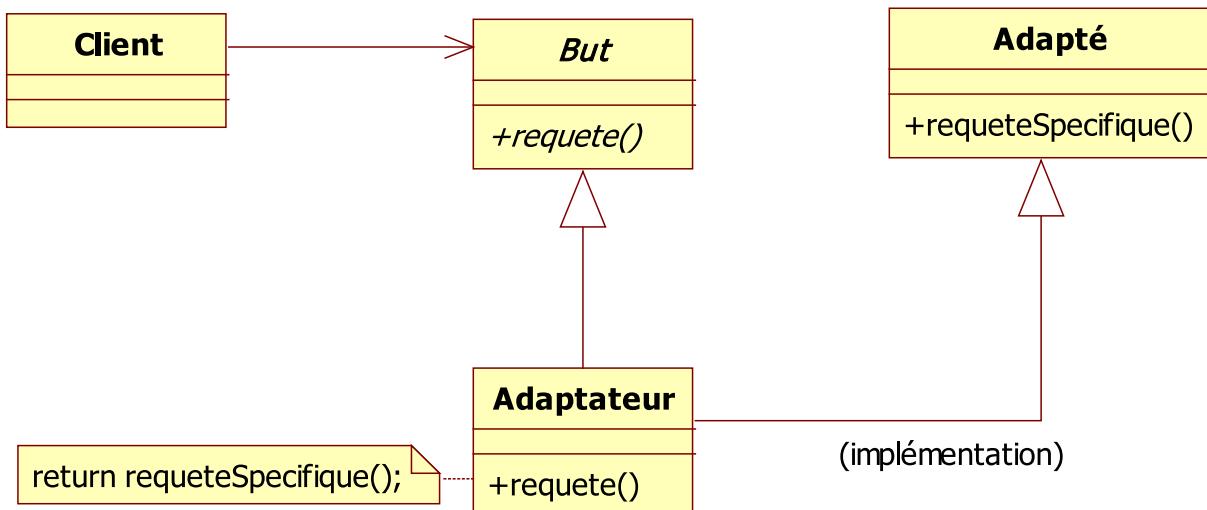


Figure 61. Modèle UML du patron Adaptateur

Composite

Composite permet de composer des objets en des structures arborescentes pour représenter des hiérarchies composant/composé. Permet au client de traiter d'une façon unique les objets et les combinaisons d'objets.

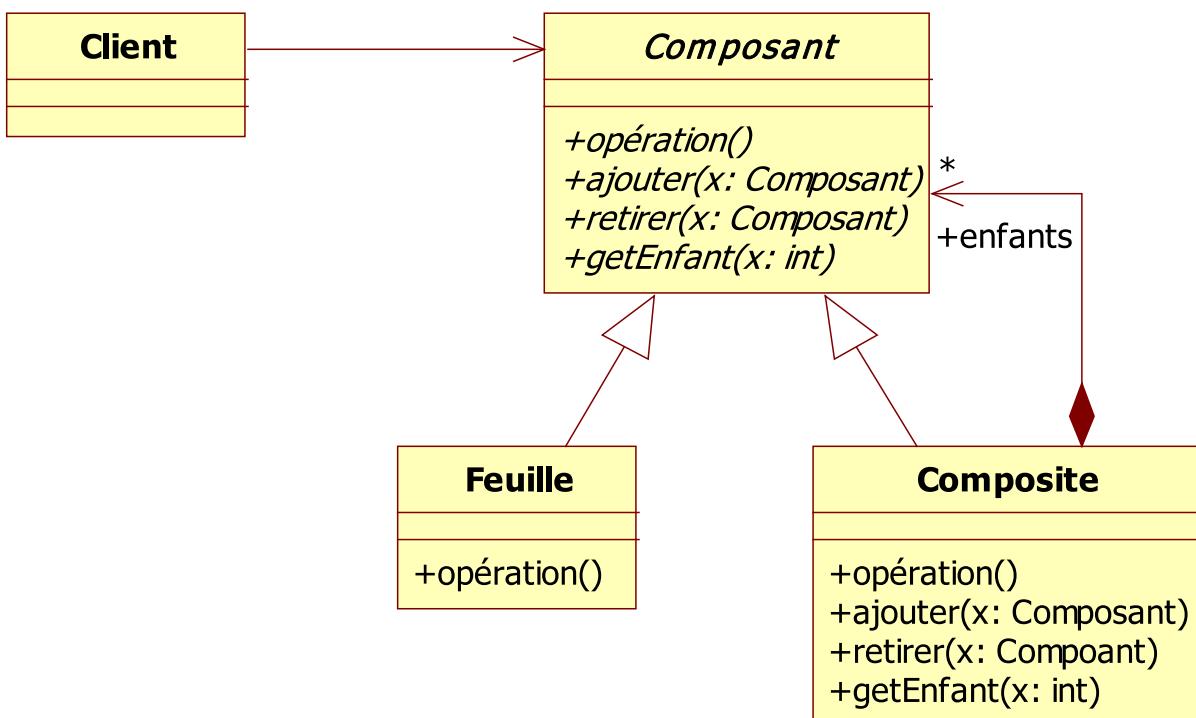


Figure 62. Modèle UML du patron Composite

Procuration (Proxy^{uk})

Procuration (Proxy) fournit à un tiers un mandataire ou un remplaçant, pour contrôler l'accès à cet objet.

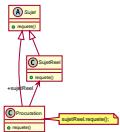


Figure 63. Modèle UML du patron Proxy

Références

- [Gof] Design Patterns: Elements of reusable object oriented software. 1994.
- [Freeman04] Design Patten - Head First. Bert Bates, Eric Freeman, Elisabeth Freeman, Kathy Sierra. O'Reilly, 09/2004.
- [Freeman05] Tête la première : Design Pattern. Bert Bates, Eric Freeman, Elisabeth Freeman, Kathy Sierra. Editions O'Reilly. 2005.
- [Cysboy] Apprenez à programmer en Java. Par [cysboy](#). Disponible [ici](#) (le 2016-11-16).
- GOPROD - De bonnes pratiques au service de la conception orientée objets. Disponible [ici](#) (le 2016-11-16).
- [Larman05] Larman, Craig. Applying UML and Patterns – An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd ed.). Prentice Hall. 2005. ISBN 0-13-148906-2.
- [Meyer88] Meyer, Bertrand. Object-Oriented Software Construction. Prentice Hall. 1988. ISBN 0-13-629049-3.
- [Martin03] “Principles Of OOD”, Robert C. Martin (“Uncle BOB”), <http://butunclebob.com> (Last verified 2014-07-17). Note the reference to “the first five principles”, though the acronym is not used in this article. Dates back to at least 2003.

About...

Document réalisé par [Jean-Michel Bruel](#) via [Asciidoctor](#) (version 1.5.5) de 'Dan Allen', lui-même basé sur [Asciidoc](#). Pour l'instant ce document est libre d'utilisation et géré par la 'Licence Creative Commons'. licence Creative Commons Paternité - Partage à l'Identique 3.0 non transposé.