Quelques Design Patterns

Conception Orientée Objet

Jean-Christophe Routier
Licence mention Informatique
Université Lille 1

Singleton

Ensure a class only has one instance, and provide a global point of access to it.

- Possible pour toute classe “sans état” ou avec état “invariable”.

Mise en œuvre :

- Rendre privé le constructeur,
- Construire une instance de la classe comme “dans” la classe,
- Fournir une méthode d’accès à cette instance

En Java

```
public class SingletonClass {
   private SingletonClass() { }
   private SingletonClass uniqueInstance = new SingletonClass();
   public static SingletonClass getTheInstance() {
      return uniqueInstance; }
}
```

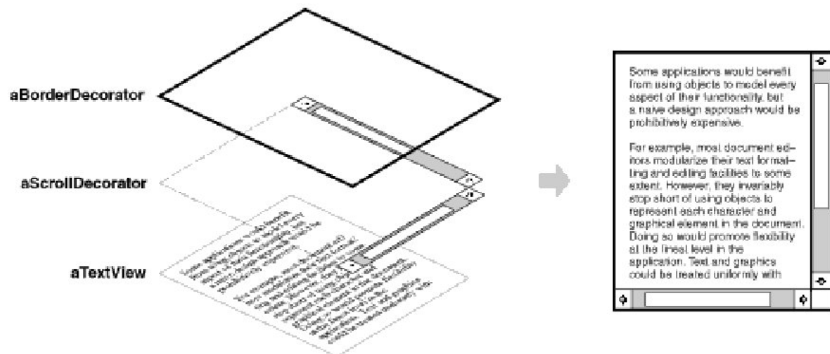
ou

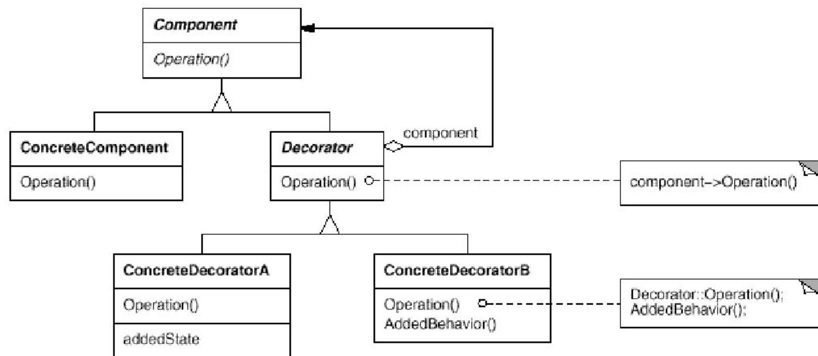
```
public class SingletonClass {
   private SingletonClass() { }
   public static final SingletonClass seuleInstance = new SingletonClass();
}
```

- Usage un peu détourné : faible nombre d'instances :
  ↪ implémentation des types énumérés en JAVA (< 1.5)

Decorator

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.





exemple : `BufferedReader` dans `java.io`

Factory Method

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

"virtual constructor"

- L'ajout de nouvelles classes se fait grâce au polymorphisme
  $\Rightarrow$ modifications au niveau des méthodes héritées
- **MAIS** il reste à créer les instances,
  $\Rightarrow$ effets aux endroits où sont créés les objets

À utiliser quand :

- une classe ne peut pas anticiper la classe des objets qu'elle doit créer
- une classe veut que ses sous-classes spécifient les objets qu'elle crée.

exemples : méthode `iterator()` dans `Collection`

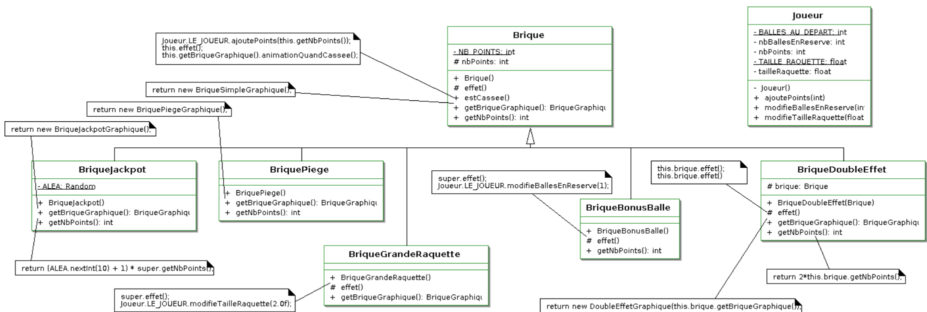
Construction gérée par les sous-classes

```
package oie;
public abstract class AbstractCase {
   (...)
   public abstract CaseGraphique getMonDessin();
}// AbstractCase

public class Oie extends AbstractCase {
   (...)
   public CaseGraphique getMonDessin(){
      return new CaseGraphiqueOie(this.index);
   }
}// Oie

public class Normale extends AbstractCase {
   (...)
   public CaseGraphique getMonDessin(){
      return new CaseGraphiqueNormale(this.index);
   }
}// Normale
```

avec

```
public interface CaseGraphique {
   public void display();
}
public class CaseGraphiqueOie implements CaseGraphique { ... }
public class CaseGraphiqueNormale implements CaseGraphique {... }
public class CaseGraphiqueAttente implements CaseGraphique { ... }
```

Un exemple connu : plusieurs design patterns



arcanoid.graphique

«interface»
BriqueGraphique
+ *animationQuandCassee(*
+ *display()*

BriqueGrandeRaquetteGraphique
+ BriqueGrandeRaquetteGraphique()
+ animationQuandCassee()
+ display()

BriqueSimpleGraphique
+ BriqueSimpleGraphique()
+ animationQuandCassee()
+ display()

BriquePiegeGraphique
+ BriquePiegeGraphique()
+ animationQuandCassee()
+ display()

BriqueJackpotGraphique
+ BriqueJackpotGraphique()
+ animationQuandCassee()
+ display()

DoubleEffetGraphique
brique: BriqueGraphique
+ DoubleEffetGraphique(BriqueGraphiqu
+ animationQuandCassee()
+ display()

Un exemple connu : plusieurs design patterns

Singleton, décorateur et factory method.

Joueur.LE_JOUEUR.ajoutePoints(this.getNbPoints());
this.effet();
this.getBriqueGraphique().animationQuandCassee();

return new BriqueSimpleGraphique();

return new BriquePiegeGraphique();

return new BriqueJackpotGraphique();

Brique
- _NB_POINTS: int_
nbPoints: int
+ Brique()
# effet()
+ estCassee()
+ getBriqueGraphique(): BriqueGraphiqu
+ getNbPoints(): int

Joueur
- _BALLES_AU_DEPART: int_
- nbBallesEnReserve: int
- nbPoints: int
- _TAILLE_RAQUETTE: float_
- tailleRaquette: float
- Joueur()
+ ajoutePoints(int)
+ modifieBallesEnReserve(in
+ modifieTailleRaquette(float

BriqueJackpot
- _ALEA: Random_
+ BriqueJackpot()
+ getBriqueGraphique(): BriqueGraphiqu
+ getNbPoints(): int

BriquePiege
+ BriquePiege()
+ getBriqueGraphique(): BriqueGraphiqu
+ getNbPoints(): int

super.effet();
Joueur.LE_JOUEUR.modifieBallesEnReserve(1);

this.brique.effet();
this.brique.effet()

BriqueDoubleEffet
brique: Brique
+ BriqueDoubleEffet(Brique)
# effet()
+ getBriqueGraphique(): BriqueGraphiqu
+ getNbPoints(): int

BriqueBonusBalle
+ BriqueBonusBalle()
# effet()
+ getNbPoints(): int

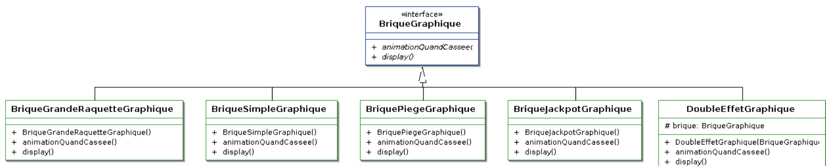
return (ALEA.nextInt(10) + 1) * super.getNbPoints();

BriqueGrandeRaquette
+ BriqueGrandeRaquette()
# effet()
+ getBriqueGraphique(): BriqueGraphiqu

return 2*this.brique.getNbPoints();

super.effet();
Joueur.LE_JOUEUR.modifieTailleRaquette(2.0f);

return new DoubleEffetGraphique(this.brique.getBriqueGraphique());



Un exemple connu : plusieurs design patterns

Singleton, décorateur et factory method.

Joueur.LE_JOUEUR.ajoutePoints(this.getNbPoints());
this.effet();
this.getBriqueGraphique().animationQuandCassee();

return new BriqueSimpleGraphique();

return new BriquePiegeGraphique();

return new BriqueJackpotGraphique();

Factory Method

Brique
- NB_POINTS: int
nbPoints: int
+ Brique()
# effet()
+ estCassee()
+ getBriqueGraphique(): BriqueGraphiqu
+ getNbPoints(): int

Joueur
- BALLES_AU_DEPART: int
- nbBallesEnReserve: int
- nbPoints: int
- TAILLE_RAQUETTE: float
- tailleRaquette: float
- Joueur()
+ ajoutePoints(int)
+ modifieBallesEnReserve(in
+ modifieTailleRaquette(float

Singleton

BriqueJackpot
- ALEA: Random
+ BriqueJackpot()
+ getBriqueGraphique(): BriqueGraphiqu
+ getNbPoints(): int

BriquePiege
+ BriquePiege()
+ getBriqueGraphique(): BriqueGraphiqu
+ getNbPoints(): int

super.effet();
Joueur.LE_JOUEUR.modifieBallesEnReserve(1);

this.brique.effet();
this.brique.effet()

BriqueDoubleEffet
brique: Brique
+ BriqueDoubleEffet(Brique)
# effet()
+ getBriqueGraphique(): BriqueGraphiqu
+ getNbPoints(): int

BriqueBonusBalle
+ BriqueBonusBalle()
# effet()
+ getNbPoints(): int

return (ALEA.nextInt(10) + 1) * super.getNbPoints();

BriqueGrandeRaquette
+ BriqueGrandeRaquette()
# effet()
+ getBriqueGraphique(): BriqueGraphiqu

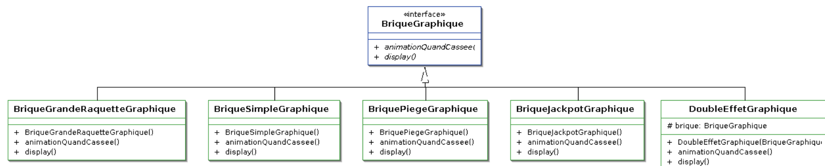
super.effet();
Joueur.LE_JOUEUR.modifieTailleRaquette(2.0f);

return 2*this.brique.getNbPoints();

return new DoubleEffetGraphique(this.brique.getBriqueGraphique());

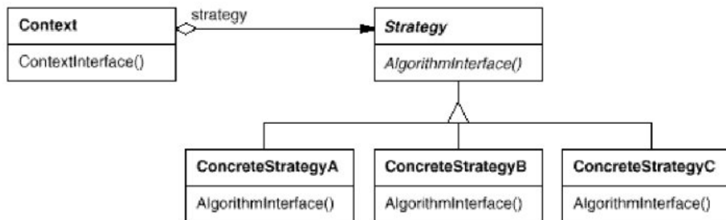
Décorateur

arcanoid.graphique



Strategy

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

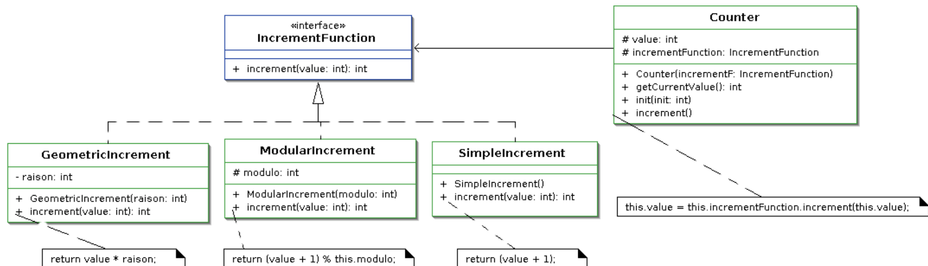


`JPanel` et `LayoutManager`

`Counter` et `IncrementFunction` ou `Joueur` et `Strategie`

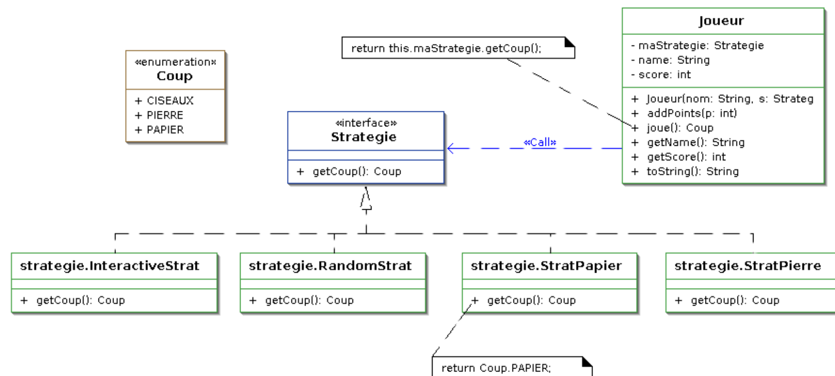
Exemple : des compteurs

```
public interface IncrementFunction {
   public int increment(int value);
}

public class SimpleIncrement implements IncrementFunction { ... }
public class ModularIncrement implements IncrementFunction { ... }
public class AnotherIncrement implements IncrementFunction { ... }

public class Counter {
   private int value;
   private IncrementFunction incrementF;
   public Counter(int value, IncrementFunction incrementF) {
      this.value = value; his.incrementF = incrementF;
   }
   public int getCurrentValue() { return value : }
   public void increment() { value = incrementF.increment(value); }
   public void initValue(int init) { this.value = init; }
}


// ... utilisation

Counter simpleCounter = new Counter(0, new SimpleIncrement());
Counter modularCounter = new Counter(0, new ModularIncrement(7));
Counter anotherCounter = new Counter(0, new AnotherIncrement());
```


«interface»
IncrementFunction
+ increment(value: int): int
Counter
value: int
incrementFunction: IncrementFunction
+ Counter(incrementF: IncrementFunction)
+ getCurrentValue(): int
+ init(init: int)
+ increment()
GeometricIncrement
- raison: int
+ GeometricIncrement(raison: int)
+ increment(value: int): int
ModularIncrement
modulo: int
+ ModularIncrement(modulo: int)
+ increment(value: int): int
SimpleIncrement
+ SimpleIncrement()
+ increment(value: int): int
this.value = this.incrementFunction.increment(this.value);
return value * raison;
return (value + 1) % this.modulo;
return (value + 1);

Exemple : stratégies de jeu

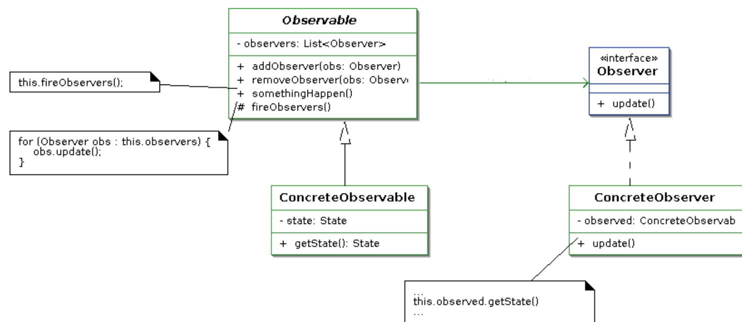
```
public interface Strategie {
   public Coup calculDuCoupAJouer(...); // classe Coup définie "ailleurs"
}

public class Joueur {
   private Strategie maStrategie;
   public Joueur(Strategie strategie, ....) {
      maStrategie = strategie; ...
   }
   public Coup joue() {
      return maStrategie.calculDuCoupAJouer(...);
   }
}

public class StrategieAleatoire implements Strategie {
   public Coup calculDuCoupAJouer(...) { ... };  // choix aléatoire
}
public class StrategieImpl implements Strategie {
   public Coup calculDuCoupAJouer(...) { ... };  // un autre calcul
}

// ... utilisation
   Joueur joueur1 = new Joueur(new StrategieAleatoire());
   Joueur joueur2 = new Joueur(new StrategieImpl());
   new Jeu(joueur1, joueur2).unTourDeJeu();
```



«enumeration»
Coup
+ CISEAUX
+ PIERRE
+ PAPIER
return this.maStrategie.getCoup();
Joueur
- maStrategie: Strategie
- name: String
- score: int
+ Joueur(nom: String, s: Strateg
+ addPoints(p: int)
+ joue(): Coup
+ getName(): String
+ getScore(): int
+ toString(): String
«interface»
Strategie
+ getCoup(): Coup
«Call»
strategie.InteractiveStrat
+ getCoup(): Coup
strategie.RandomStrat
+ getCoup(): Coup
strategie.StratPapier
+ getCoup(): Coup
strategie.StratPierre
+ getCoup(): Coup
return Coup.PAPIER;

Observer/Observable

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.



- Observer/Observable ou Abonneur/Abonné ou “event generator” idiom

Problème

Un objet (acteur) doit réagir lorsqu'un autre objet remarque un événement, mais ce dernier n'a pas nécessairement conscience de l'existence de l'acteur (en fait **des** acteurs).

Les (ré)acteurs peuvent :

- être de différentes natures
- changer sans que l'émetteur de l'événement en soit conscient

Exemple :

- la gestion des événements dans awt/Swing

Les contraintes

- Plusieurs objets peuvent être avertis des événements émis par une source
- Le nombre et la nature des objets avertis ne sont potentiellement pas connus à la compilation et peuvent changer dans le temps.
- L'émetteur de l'événement et le récepteur ne sont pas fortement liés.

⇒ Mettre en place un mécanisme de délégation entre l'émetteur de l'événement (*event generator*) et le récepteur (*listener*)

Scénario **(d'après B. Venners)**

Un téléphone sonne, différentes personnes ou machines peuvent potentiellement être concernées par l'appel. Les personnes peuvent sortir ou entrer dans la pièce et ainsi sortir ou entrer du "champ d'intéressement" du téléphone.

Les étapes de design en Java

1. Définir les classes d'événements
2. Définir les interfaces des `listeners`
   - Définir les classes des `adapters` (optionnel)
3. Définir la classe émettrice (génératrice des événements)
4. Définir les classes réceptrices (les `listeners`)

Étape 1 : définir les classes d'événements

- Définir une classe par type d'événements qui peuvent être émis par le générateur d'événements
- Faire hériter ces classes de `java.util.EventObject`
- Concevoir les classes d'événements de telle sorte qu'un événement englobe toute l'information qui doit être transmise à l'observateur
- Donner à une classe d'évenement un nom de la forme `XXXEvent`

```
package essais.observer;
public class TelephoneEvent extends java.util.EventObject {
   public TelephoneEvent(Telephone source) {
      super(source);
   }
}
```

- `EventObject.getSource()` : disposer de la source permet d'être abonné à plusieurs sources pour le même type d'évenements
- 2 modèles dans le design pattern “observer” :
  - **pull-model** (l'observateur doit extraire les infos de la source)
  - **push-model** (toute l'information est encapsulée directement dans l'événement)

Étape 2 : définir les interfaces des `listeners`

- pour chaque type d'événements, définir une interface qui hérite de `java.util.EventListener` et contient, pour chaque événement, une déclaration de méthode qui sera déclenchée lors de la notification de l'occurrence d'un événement
- Le nom de cette interface est obtenu à partir du nom de la classe de l'événement en remplaçant `Event` par `Listener`
- Les noms des méthodes de l'interface sont construits à partir d'un verbe au passé indiquant la situation d'activation.
- Chaque méthode retourne `void` et prend un paramètre qui est de la classe de l'événement

```
package essais.observer;
public interface TelephoneListener extends java.util.EventListener {
   public void telephoneRang(TelephoneEvent e);
   public void telephoneAnswered(TelephoneEvent e);
}
```

Étape 2 (option) : définir les classes des adapters

Etape optionnelle

- Pour chaque `listener` on définit une classe qui l'implémente en définissant des méthodes creuses
- La classe de l'`adapter` est obtenue en remplaçant `Listener` par `Adapter`.

```
package essais.observer;
public class TelephoneAdapter implements TelephoneListener {
   public void telephoneRang(TelephoneEvent e) { }
   public void telephoneAnswered(TelephoneEvent e) { }
}
```

Étape 3 : définir la classe émettrice

C'est la classe génératrice des événements.

- Pour chaque type d'événements émis, définir un couple de méthodes add/remove
- Le nom de ces méthodes est de la forme :
  `add`*listener-interface-name*`()` et `remove`*listener-interface-name*`()`.
  Méthode d'*abonnement*, *désabonnement*
- Pour chacune des méthodes des interfaces des `listeners`, définir une méthode `private` de propagation de l'événement.
- Cette méthode est nommée : `fire`*listener-method-name*.
- Déclencher les événements.

```
package essais.observer;
import java.util.*;
public class Telephone {
  private ArrayList<TelephoneListener> telephoneListeners = new ArrayList<TelephoneListener>();
  public void ringPhone() { fireTelephoneRang(); }
  public void answerPhone() { fireTelephoneAnswered(); }
  public synchronized void addTelephoneListener(TelephoneListener l) {
    if (telephoneListeners.contains(l)) { return; }
    telephoneListeners.add(l);
  }
  public synchronized void removeTelephoneListener(TelephoneListener l){
    telephoneListeners.remove(l);
  }
  private void fireTelephoneRang() {
    ArrayLis<TelephoneListener>t tl = (ArrayList<TelephoneListener>) telephoneListeners.clone();
    if (tl.size() == 0) { return; }
    TelephoneEvent event = new TelephoneEvent(this);
    for (TelephoneListener listener : tl) {
      listener.telephoneRang(event);
    }
  }
  private void fireTelephoneAnswered() { ... }
}
```

Étape 4 : définir les classes réceptrices

- Les classes qui veulent être des `listeners` n'ont qu'à implémenter l'interface `listener` associée

```
package essais.observer;
public class AnsweringMachine implements TelephoneListener {
   public void telephoneRang(TelephoneEvent e) {
      System.out.println("AM hears the phone ringing.");
   }
   public void telephoneAnswered(TelephoneEvent e) {
      System.out.println("AM sees that the phone was answered.");
   }
}

package essais.observer;
class MyTelephoneAdapter extends TelephoneAdapter {
   public void telephoneRang(TelephoneEvent e) {
      System.out.println("I'll get it!");
   }
}
public class Person {
   public void listenToPhone(Telephone t) {
      t.addTelephoneListener(new MyTelephoneAdapter());
   }
}
```

Exploitation

```
package essais.observer;

public class Example1 {
  public static void main(String[] args) {
    Telephone ph = new Telephone();   // émetteur
    AnsweringMachine am = new AnsweringMachine(); //listener
    Person bob = new Person();        // ``possède'' un listener

    ph.addTelephoneListener(am);      // abonnement
    bob.listenToPhone(ph);            // abonnement

    ph.ringPhone();                   // provoque l'émission
    ph.answerPhone();                 //  d'un événement
  }
}
```

Règles d'implémentation

- passer la même instance à tous les listeners pour un événement donné
- rendre l'objet événement immuable
- utiliser un seul thread pour avertir tous les listeners
- créer une image de la liste des listeners abonnés avant de les avertir
- les méthodes des listeners doivent être éxécutées rapidement (car notifiés 1 par 1)
- l'ordre des notifications ne doit pas être important
- créer des `adpaters` pour permettre aux listeners de se concentrer que sur certains événements.

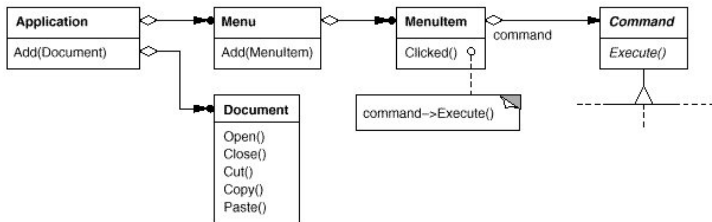
- différences de fréquences entre l'occurrence de deux évenements émis par une même source : créer des listeners différents
  cf. `MouseListener` et `MouseMotionListener`
- `java.util.Observer` et `java.util.Observable`

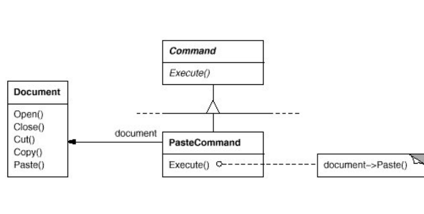
Command

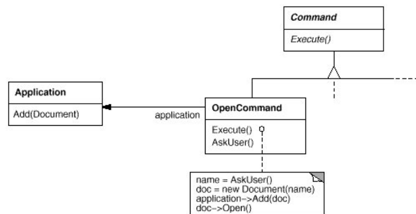
Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Client
Invoker
Command
Execute()
Receiver
Action()
receiver
ConcreteCommand
Execute()
state
receiver–>Action();

Exemple : gestion d'un menu







```
public interface Command {
   public boolean execute();
}

public class Menu {
   private HashMap<String,Command> commands = new HashMap<String,Command>();
   public Menu (){
      init();
   }
   private void init() {
      commands.put("save",new SaveCommand());
      commands.put("load",new LoadCommand());
      commands.put("help",new HelpCommand());
      (...) }
   public void act() {                                  // très allégé ici
      System.out.println("what do you want to do (typing 'help' can use)?");
      command = Input.readString();
      commands.get(command).execute();
   }
}
public class SaveCommand implements Command {
   public void execute() { (...) }
}
public class LoadCommand implements Command { public void execute() { (...) } }
public class HelpCommand implements Command { public void execute() { (...) } }
```

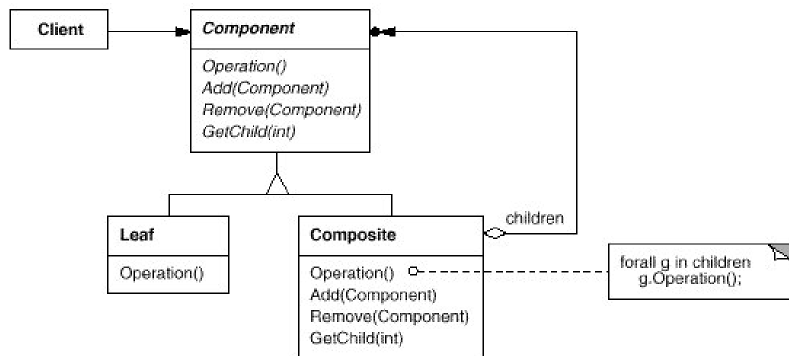
`commands.put("save",new SaveCommand());` : Dépendances ? Inversion of Control ?

Dépendances inversées

```
public interface Command {
   public boolean execute();
   public String getHelpMsg();
}
public class Menu implements Command {
   private HashMap<String,Command> commands = new HashMap<String,Command>();
   public Menu (){
      this.addCommand("help",this);
   }
   private void addCommand(String key, Command command) {     // inversion du contrôle
      commands.put(key,command);
   }
   public void act() {
      System.out.println("what do you want to do (typing 'help' can use)?");
      commands.get(Input.readString()).execute();
   }
   public void execute() { // itère sur les clefs et appelle getHelpMsg sur les valeurs
   }
   public String getHelpMsg() { return "affiche cette aide"; }
}
public class SaveCommand implements Command {
   public void execute() { (...) }
   public String getHelpMsg() { return "sauvegarde ..." }
}
public class LoadCommand implements Command {
   public void execute() { (...) }
   public String getHelpMsg() { return "chargement ..." }
}
```

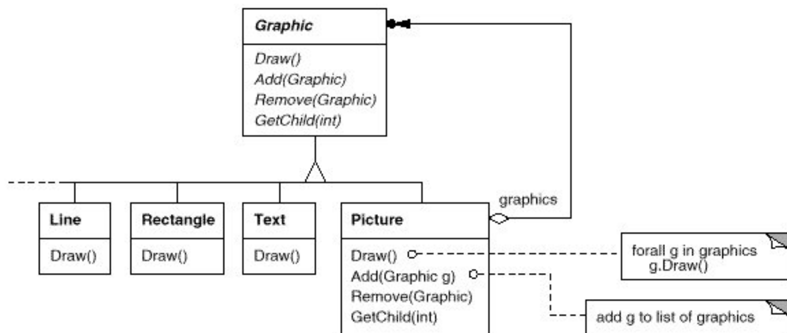
Composite

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.



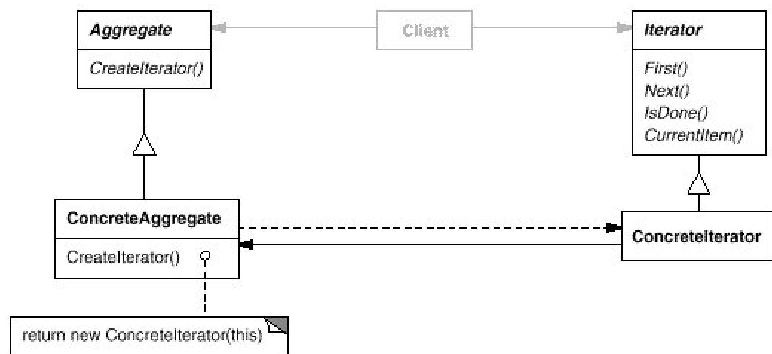
Exemple

composants AWT/SWING et leur méthode `repaint()`



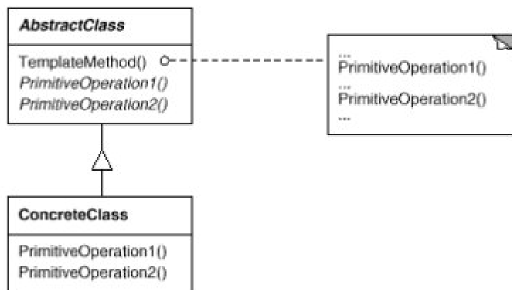
Iterator

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

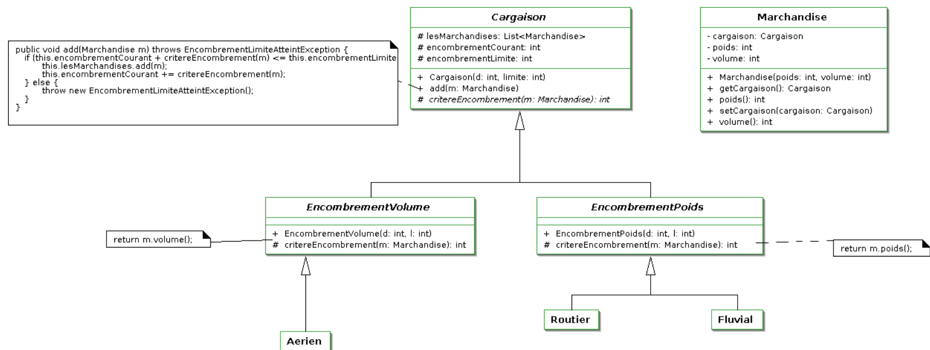


Template method

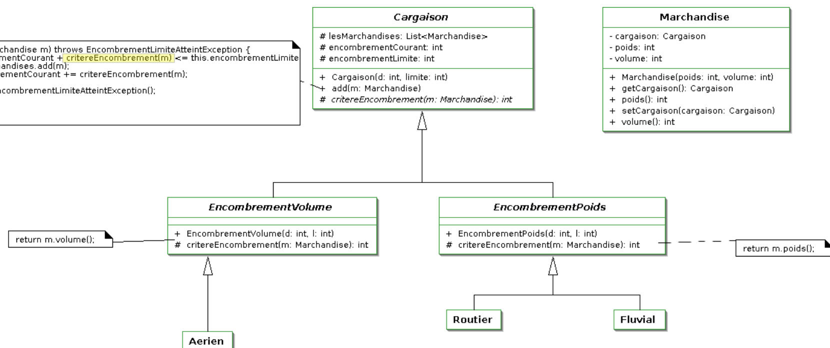
Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.



Exemple : algorithme MinMAX



public void add(Marchandise m) throws EncombrementLimiteAtteintException {
if (this.encombrementCourant + critereEncombrement(m) <= this.encombrementLimite
this.lesMarchandises.add(m);
this.encombrementCourant += critereEncombrement(m);
} else {
throw new EncombrementLimiteAtteintException();
}
}
Cargaison
lesMarchandises: List<Marchandise>
encombrementCourant: int
encombrementLimite: int
+ Cargaison(d: int, limite: int)
+ add(m: Marchandise)
# critereEncombrement(m: Marchandise): int
Marchandise
- cargaison: Cargaison
- poids: int
- volume: int
+ Marchandise(poids: int, volume: int)
+ getCargaison(): Cargaison
+ poids(): int
+ setCargaison(cargaison: Cargaison)
+ volume(): int
return m.volume();
EncombrementVolume
+ EncombrementVolume(d: int, l: int)
# critereEncombrement(m: Marchandise): int
EncombrementPoids
+ EncombrementPoids(d: int, l: int)
# critereEncombrement(m: Marchandise): int
return m.poids();
Aerien
Routier
Fluvial


public void add(Marchandise m) throws EncombrementLimiteAtteintException {
if (this.encombrementCourant + critereEncombrement(m) <= this.encombrementLimite
this.lesMarchandises.add(m);
this.encombrementCourant += critereEncombrement(m);
} else {
throw new EncombrementLimiteAtteintException();
}
}
Cargaison
lesMarchandises: List<Marchandise>
encombrementCourant: int
encombrementLimite: int
+ Cargaison(d: int, limite: int)
+ add(m: Marchandise)
# critereEncombrement(m: Marchandise): int
Marchandise
- cargaison: Cargaison
- poids: int
- volume: int
+ Marchandise(poids: int, volume: int)
+ getCargaison(): Cargaison
+ poids(): int
+ setCargaison(cargaison: Cargaison)
+ volume(): int
EncombrementVolume
+ EncombrementVolume(d: int, l: int)
# critereEncombrement(m: Marchandise): int
return m.volume();
EncombrementPoids
+ EncombrementPoids(d: int, l: int)
# critereEncombrement(m: Marchandise): int
return m.poids();
Aerien
Routier
Fluvial

Retour sur “factory method”

Exemples.

```
public abstract class Shape {
   public abstract void draw() ;
   public abstract void erase() ;
   public static Shape factory(String type) throws BadShapeCreation {
      if(type.equals("Circle")) return new Circle() ;
      if(type.equals("Square")) return new Square() ;
      throw new BadShapeCreation(type) ;
   }
}

public class Circle extends Shape {
   public Circle() {} // Friendly constructor
   public void draw() { System.out.println("Circle.draw") ; }
   public void erase() { System.out.println("Circle.erase") ; }
}
```

OCP ?

Exemple (suite)

```
public interface ShapeFactory {
  public Shape createShape();
}
public class AllShapeFactories {
  public static Map<String,ShapeFactory> factories = new HashMap<String,ShapeFactory>();
  public static final Shape createShape(String id) {// "throw BadShapeCreation"
    return factories.get(id).createShape();
  }
public class Circle extends Shape {
  static { ShapeFactory.factories.put("Circle", new CircleFactory()); }
  (...)
}
public class CircleFactory implements ShapeFactory {
  public Shape createShape() { return new Circle(); }
}
public class Rectangle extends Shape {
  static { ShapeFactory.factories.put("Rectangle", new RectangleFactory()); }
  (...)
}
public class RectangleFactory implements ShapeFactory {
  public Shape createShape() { return new Rectangle(); }
}
```

Usage

```
      //création à partir du nom de classe
Shape shape = AllShapeFactories.createShape("Circle");
...
      // création "aléatoire"
int alea = new random.nextInt(ShapeFactory.factories.size());
Iterator<String> it = AllShapeFactories.factories.keySet().iterator();
for (int i = 0; i < alea; i++) { it.next(); }
shape = ShapeFactory.createShape(it.next());
```

- éviter la “pollution des classes” : utiliser des classe internes

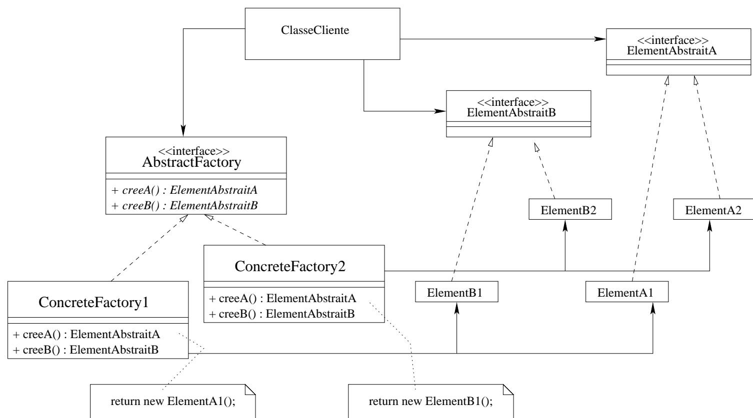
```
public class Circle extends Shape {
  static { ShapeFactory.factories.put("Circle", new Circle.MaFactory()); }
  (...)
  class MaFactory implements ShapeFactory {          // classe interne
    public Shape createShape() { return new Circle(); }
  }
}
public class Rectangle extends Shape {
  static { ShapeFactory.factories.put("Rectangle", new Rectangle.MaFactory()); }
  class MaFactory implements Factory {
    public Shape createShape() { return new Rectangle(); }
  }
  (...)
}
```

Avec chargement dynamique...

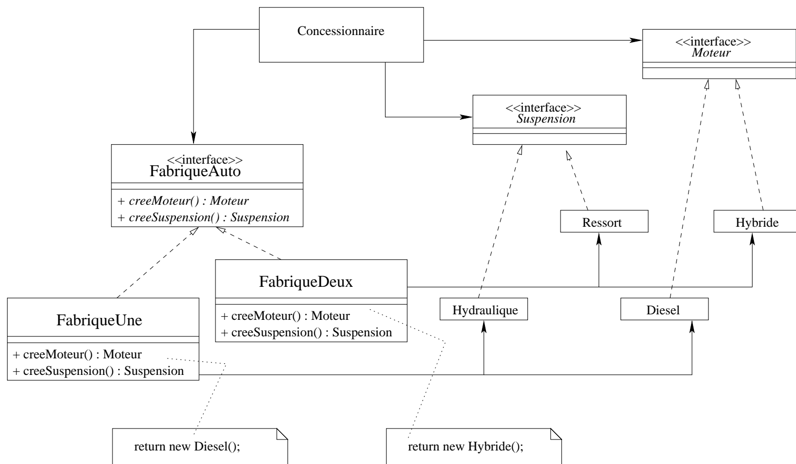
```
public interface Shape {
   public void draw();
   public void erase();
}
public abstract class ShapeFactory {
   protected abstract Shape create();
   static Map<String,ShapeFactory> factories = new HashMap<String,ShapeFactory>();
   // A Template Method :
   public static final Shape createShape(String id) throws BadShapeCreation {
   if(!factories.containsKey(id)) {
      try {
         Class.forName(id);                                 // Chargement dynamique
      } catch(ClassNotFoundException e) { throw new BadShapeCreation(id); }
      // Vérifie si elle a été ajoutée
      if(!factories.containsKey(id)) throw new BadShapeCreation(id);
   }
   return factories.get(id).create();
   }
}
public class Circle implements Shape {
   private Circle() {}
   public void draw() { System.out.println("Circle.draw"); }
   public void erase() { System.out.println("Circle.erase"); }
   private static class Factory extends ShapeFactory {
      protected Shape create() { return new Circle(); }
   }
   static { ShapeFactory.factories.put("Circle", new Circle.Factory()); }
}
```

Abstract Factory

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.



Exemple



Visitor

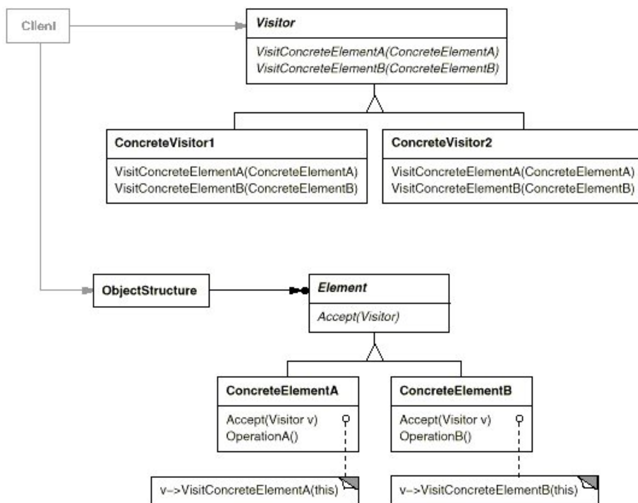
Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

- Séparer la structure d'une collection des traitements effectués sur la collection
- En combinaison avec le design pattern *Composite*

On dispose de 2 hiérarchies distinctes :

- la hiérarchie des *données*
- la hiérarchie des *traitements* (visiteur)

Permet d'"ajouter des méthodes" dans la hiérarchie des données (sans la modifier).


Client
Visitor
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor1
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor2
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ObjectStructure
Element
Accept(Visitor)
ConcreteElementA
Accept(Visitor v)
OperationA()
ConcreteElementB
Accept(Visitor v)
OperationB()
v->VisitConcreteElementA(this)
v->VisitConcreteElementB(this)

Exemple : affichage de collections.

- JAVA : utilisation des `iterator/Iterable` ?
- Problème : modification des types des objets contenus dans la collection...

```
public void messyPrintCollection(Collection collection) {
   for (Object o : collection) {
      System.out.println(o.toString());
   }
}
```

Introduction du traitement de collections de collections...

(from J. Blosser)

```
public void messyPrintCollection(Collection collection) {
   for (Object o : collection) {
      if (o instanceof Collection)
         messyPrintCollection((Collection)o);
      else
         System.out.println(o.toString());
   }
}
```

Ajout d'autres modifications de traitements...

```
public void messyPrintCollection(Collection collection) {
   for (Object o : collection) {
      if (o instanceof Collection)
         messyPrintCollection((Collection)o);
      else if (o instanceof String)
         System.out.println("'"+o.toString()+"'");
      else if (o instanceof Float)
         System.out.println(o.toString()+"f");
      else
         System.out.println(o.toString());
   }
}
```

Open Close Principle ?!?

Visitor à la rescousse

```
public interface Visitor {
   public void visitCollection(Collection<Visitable> collection);
   public void visitString(String string);
   public void visitFloat(Float float);
}

public interface Visitable {
   public void accept(Visitor visitor);
}

public class VisitableString implements Visitable {
   private String value;
   public VisitableString(String string) { value = string; }
   public void accept(Visitor visitor) {
      visitor.visitString(this.value);
   }
}
```

+ VisitableFloat et VisitableCollection...

et le *visiteur* qui va avec...

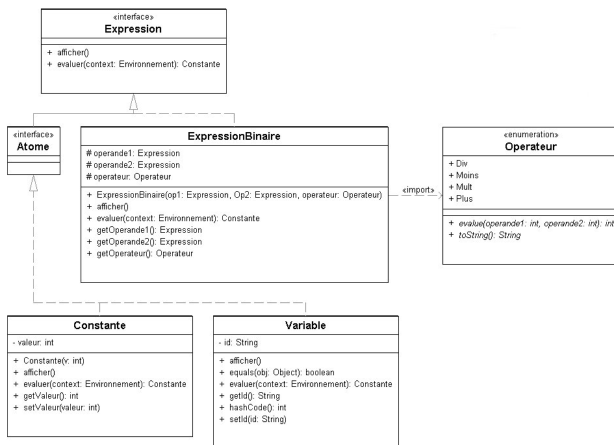
```
public class PrintVisitor implements Visitor {
   public void visitCollection(Collection<Visitable> collection) {
      for (Visitable v : collection) {
            v.accept(this);
      }
   }

   public void visitString(String string) {
      System.out.println("'"+string+"'");
   }

   public void visitFloat(Float float) {
      System.out.println(float.toString()+"f");
   }
}
```

"double dispatch" : `Visitor`→`Visitable`→`Visitor`

Bof !

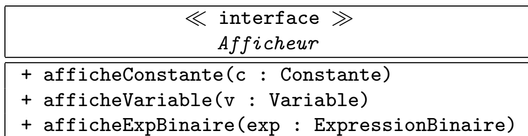
- plus de `if-then-else`
- mais... beaucoup de code + “enrobage” des objets pour l’interface `Visitable`
- de plus : ajout de `VisitableInteger`
  $\Rightarrow$ changement de l’interface `Visitor` !
- Solution ? utilisation de la réflexivité (voir plus tard `ReflectiveVisitor`)

Application : affichage d'expressions



- Affichage infixe, postfixe, préfixe...

- le visiteur :



- dans `Expression`, la méthode d'acceptation est `afficher`, qui devient :

```
public void afficher(Afficheur visitor)
```

- On crée les afficheurs :

```
public class AfficheurInfixe implements Afficheur {
  ...
}
public class AfficheurPrefixe implements Afficheur {
  ...
}
```

```
public class AfficheurInfixe implements Afficheur {
   public void afficheVariable(Variable v) { S.o.p(v.getValeur()); }
   public void afficheConstante(Constante c) { S.o.p(c.getId()); }
   public void afficheExpBinaire(ExpressionBinaire exp) {
      exp.getOperande1().afficher(this);      // afficher=≡accept
      S.o.p(exp.getOperateur());
      exp.getOperande2().afficher(this);
}
```

```
public class AfficheurPrefixe implements Afficheur {
   ...
   public void afficheExpBinaire(ExpressionBinaire exp) {
      S.o.p(exp.getOperateur());
      exp.getOperande1().afficher(this);
      exp.getOperande2().afficher(this);
}
```

```
public class AfficheurInfixeParenthese extends AfficheurInfixe {
   public void afficheExpBinaire(ExpressionBinaire exp) {
      S.o.p(exp.getOperateur("("));
      super.afficheExpBinaire(exp);
      S.o.p(exp.getOperateur(")"));
}
```

ReflectiveVisitor

```
public interface ReflectiveVisitor {
  public void visit(Object o);
}

public class PrintVisitor implements ReflectiveVisitor {
  public void visit(Collection collection) { idem }
  public void visit(String string) { idem }
  public void visit(Float float){ idem }
  public void defaultVisit(Object o) { S.o.p(o.toString()); }

  public void visit(Object o) {
    try {
      Method m = getClass().getMethod("visit",
                                      new Class[] { o.getClass() });
      m.invoke(this, new Object[] { o });
    } catch (NoSuchMethodException e) { this.defaultVisit(o); }
  }
}
```

Bénéfices ?

- Plus besoin de classes `Visitable` d'enrobage
- Possibilité de gérer les cas non pris en compte (directement) par capture d'exception :
  essayer les superclasses et interfaces (pour les arguments)...

Une "getMethod" adaptée

```
protected Method getMethod(Class c) {
   Class newc = c;
   Method m = null;
   while (m == null && newc != Object.class) {     // Try the superclasses
      try {
         m = getClass().getMethod("visit", new Class[] { newc });
      } catch (NoSuchMethodException e) {
         newc = newc.getSuperclass();
      }
   }
   if (newc == Object.class) {                       // Try the interfaces.
      Class[] interfaces = c.getInterfaces();
      for (int i = 0; i < interfaces.length; i++) {
         try {
            m = getClass().getMethod("visit", new Class[] { interfaces[i] });
         } catch (NoSuchMethodException e) {}
      }
   }
   if (m == null) {
      try {
         m = thisclass.getMethod("defaultVisit", new Class[] { Object.class });
      } catch (Exception e) { }                      // Can't happen
   }
   return m;
}
```

visit devient...

```
public void visit(Object o) {
   try {
      Method m = getMethod(o.getClass()});
      m.invoke(this, new Object[] { o });
   } catch (NoSuchMethodException e) { }
}
```

- Il est possible de garder l'interface `Visitable` pour permettre aux objets visités de contrôler la navigation si on le désire.
- pour la même collection, on peut facilement avoir différents `Visitors`