Design Patterns

O. Boissier, G. Picard
SMA/G2I/ENS Mines Saint-Etienne
Olivier.Boissier@emse.fr, Gauthier.Picard@emse.fr
Septembre 2009

Sommaire

- Introduction
- Design Patterns de création
- Design Patterns de structure
- Design Patterns de comportement
- Usage et synthèse
- Bibliographie

Introduction

Objectifs

- Modularité
  - Facilité de gestion (technologie objet)
- Cohésion
  - Degré avec lequel les tâches réalisées par un seul module sont fonctionnellement reliées
  - Une forte cohésion est une bonne qualité
- Couplage
  - Degré d'interaction entre les modules dans le système
  - Un couplage"lâche'' est une bonne qualité
- Réutilisabilité
  - Bibliothèques, frameworks (cadres)

Cohésion : "mauvais" exemple

```
public class GameBoard {

public GamePiece[ ][ ] getState() { … }
 // Méthode copiant la grille dans un tableau temporaire, résultat de l'appel de la méthode.

public Player isWinner() { … }
 // vérifie l'état du jeu pour savoir s'il existe un gagnant, dont la référence est retournée.
 // Null est retourné si aucun gagnant.

public boolean isTie() { … }
 //retourne true si aucun déplacement ne peut être effectué, false sinon.

public void display () { … }
// affichage du contenu du jeu. Espaces blancs affichés pour chacune des
// références nulles.
}
```

GameBoard est responsable des règles du jeu et de l'affichage

Cohésion : "bon" exemple

```
public class GameBoard {

public GamePiece[ ][ ] getState() { … }

public Player isWinner() { … }

public boolean isTie() { … }

}

public class BoardDisplay {

public void displayBoard (GameBoard gb) { … }
// affichage du contenu du jeu. Espaces blancs affichés pour chacune des
// références nulles.
}
```

Couplage : exemple

```
void initArray(int[] iGradeArray, int nStudents) {
   int i;
   for (i = 0; i < nStudents; i++) {
      iGradeArray[i] = 0;
   }
}
```

Couplage entre client et initArray par le Paramètre "nStudents"

```
void initArray(int[ ] iGradeArray) {
   int i;
   for (i=0; i < iGradeArray.length; i++)  {
      iGradeArray[i] = 0;
   }
}
```

Couplage faible (et meilleure fiabilité) au travers de l'utilisation de l'attribut "length"

Principes de conception (1)

- Programmer une interface plus qu'une implémentation
- Utiliser des classes abstraites (interfaces en Java) pour définir des interfaces communes à un ensemble de classes
- Déclarer les paramètres comme instances de la classe abstraite plutôt que comme instances de classes particulières

Ainsi :

⇨ les classes clients ou les objets sont indépendants des classes des objets qu'ils utilisent aussi longtemps que les objets respectent l'interface qu'ils attendent

⇨ les classes clients ou les objets sont indépendants des classes qui implémentent l'interface

Principes de conception (1) : Exemple

```
class StudentRecord {
   private Name lastName;
   private Name firstName;
   private long ID;
   public Name getLastName() { return lastName; }
   … // etc
}
```

```
class SortedList {
   Object[] sortedData = new Object[size];
   public void add(StudentRecord X) {
        // …
        Name a = X.getLastName();
        Name b = sortedData[k].getLastName();
        if (a.lessThan(b)) … else … //do something
   }
… }
```

Principes de conception (1) : Exemple Solution 1

```
class StudentRecord {
   private Name lastName;
   private Name firstName;
   private long ID;
   public boolean lessThan(Object X) {
        return lastName.lessThan(X.lastName);}
   … // etc
}
```

```
class SortedList {
   Object[] sortedData = new Object[size];
   public void add(StudentRecord X) {
        // …
        if (X.lessThan(sortedData[k])) … else … //do something
   }
… }
```

Principes de conception (1) : Exemple Solution 2

```
Interface Comparable {
   public boolean lessThan(Object X);
   public boolean greaterThan(Object X);
   public boolean equal(Object X);
}
```

```
class StudentRecord implements Comparable {
   private Name lastName;
   private Name firstName;
   private long ID;
   public boolean lessThan(Object X) {
         return lastName.lessThan(((StudentRecord)X).lastName);}
   … // etc
}
```

```
class SortedList {
   Object[] sortedData = new Object[size];
   public void add(Comparable X) {
         // …
         if (X.lessThan(sortedData[k])) … else … //do something
   } … }
```

Principes de conception (2)

- Préférer la composition d'objet à l'héritage de classes

Ainsi :

⇨ le comportement peut changer en cours d'exécution

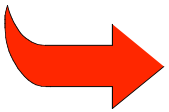
⇨ les classes sont plus focalisées sur une tâche

⇨ réduction des dépendances d'implémentation

Définition : Pattern

- Un patron décrit à la fois un **problème** qui se produit très fréquemment dans l'environnement et l'architecture de la **solution** à ce problème de telle façon que l'on puisse **utiliser** cette solution des milliers de fois sans jamais l'**adapter** deux fois de la même manière.

C. Alexander



Décrire avec succès des types de **solutions récurrentes** à des problèmes **communs** dans des types de situations

Définition : Design Pattern

- **Coad** [Coad92]
  - **Une abstraction d'un doublet, triplet ou d'un ensemble de classes** qui peut être réutilisé encore et encore pour le développement d'applications
- **Appleton** [Appleton97]
  - Une règle tripartite exprimant une relation entre un certain contexte, un certain problème qui apparaît répétitivement dans ce contexte et une certaine **configuration logicielle** qui permet la résolution de ce problème
- **Aarsten** [Aarsten96]
  - **Un groupe d'objets coopérants liés par des relations et des règles** qui expriment les liens entre un contexte, un problème de conception et sa solution

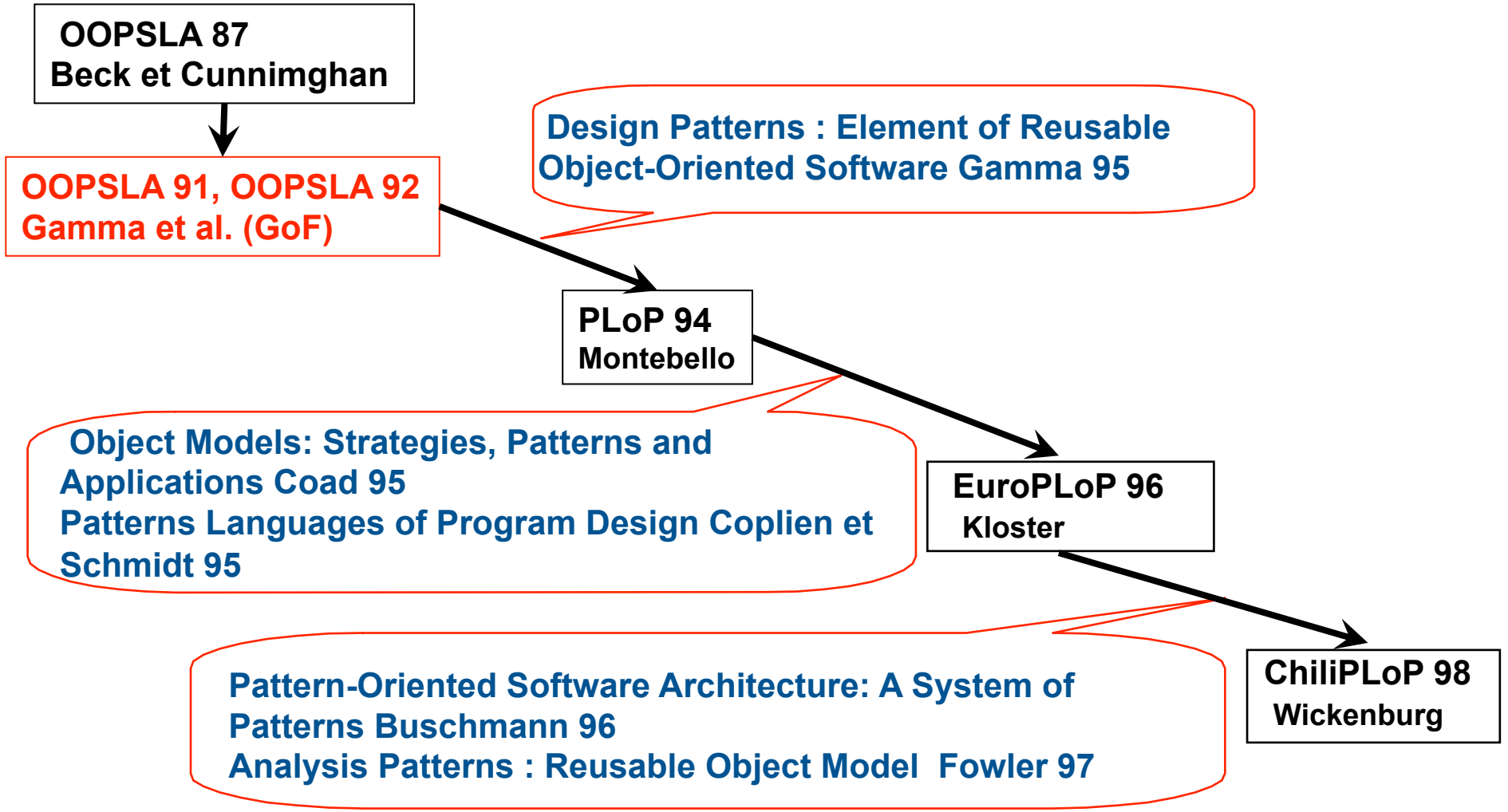
Les patrons sont des composants **logiques** décrits indépendamment d'un langage donné (solution exprimée par des modèles semi-formels)

Définition : Design Pattern (suite)

- Documentation d'une expérience éprouvée de conception
- Identification et spécification d 'abstractions qui sont au dessus du niveau des simples classes, instances
- Vocabulaire commun et aide à la compréhension de principes de conception
- Moyen de documentation de logiciels
- Aide à la construction de logiciels répondant à des propriétés précises, de logiciels complexes et hétérogènes

- Traductions : patrons de conception, schémas de conception

Historique


OOPSLA 87
Beck et Cunnimghan
OOPSLA 91, OOPSLA 92
Gamma et al. (GoF)
Design Patterns : Element of Reusable Object-Oriented Software Gamma 95
PLoP 94
Montebello
Object Models: Strategies, Patterns and Applications Coad 95
Patterns Languages of Program Design Coplien et Schmidt 95
EuroPLoP 96
Kloster
Pattern-Oriented Software Architecture: A System of Patterns Buschmann 96
Analysis Patterns : Reusable Object Model Fowler 97
ChiliPLoP 98
Wickenburg

Catégories de Patterns

- Architectural Patterns
  - schémas d'organisation structurelle de logiciels (pipes, filters, brokers, blackboard, MVC, …)
- Design Patterns
  - caractéristiques clés d'une structure de conception commune à plusieurs applications,
  - Portée plus limitée que les « architectural patterns »
- Idioms ou coding patterns
  - solution liée à un langage particulier
- Anti-patterns
  - mauvaise solution ou comment sortir d 'une mauvaise solution
- Organizational patterns
  - Schémas d'organisation de tout ce qui entoure le développement d'un logiciel (humains)

Catégories de Design Patterns

- **Création**
  - Description de la manière dont un objet ou un ensemble d'objets peuvent être créés, initialisés, et configurés
  - Isolation du code relatif à la création, à l'initialisation afin de rendre l'application indépendante de ces aspects
- **Structure**
  - Description de la manière dont doivent être connectés des objets de l'application afin de rendre ces connections indépendantes des évolutions futures de l'application
  - Découplage de l'interface et de l'implémentation de classes et d'objets
- **Comportement**
  - Description de comportements d'interaction entre objets
  - Gestion des interactions dynamiques entre des classes et des objets

Portée des Design Patterns

- Portée de Classe
  - Focalisation sur les relations entre classes et leurs sous-classes
  - Réutilisation par héritage
- Portée d'Instance
  - Focalisation sur les relations entre les objets
  - Réutilisation par composition

Design Patterns du GoF

(Gamma, Helm, Johnson, Vlissides)

		Catégorie		
		Création	**Structure**	**Comportement**
Portée	**Classe**	Factory Method	Adapter	Interpreter
				Template Method
	Objet	Abstract Factory	Adapter	Chain of Responsibility
		Builder	Bridge	Command
		Prototype	Composite	Iterator
		Singleton	Decorator	Mediator
			Facade	Memento
			Flyweight	Observer
			Proxy	State
				Strategy
				Visitor

Présentation d'un Design Pattern

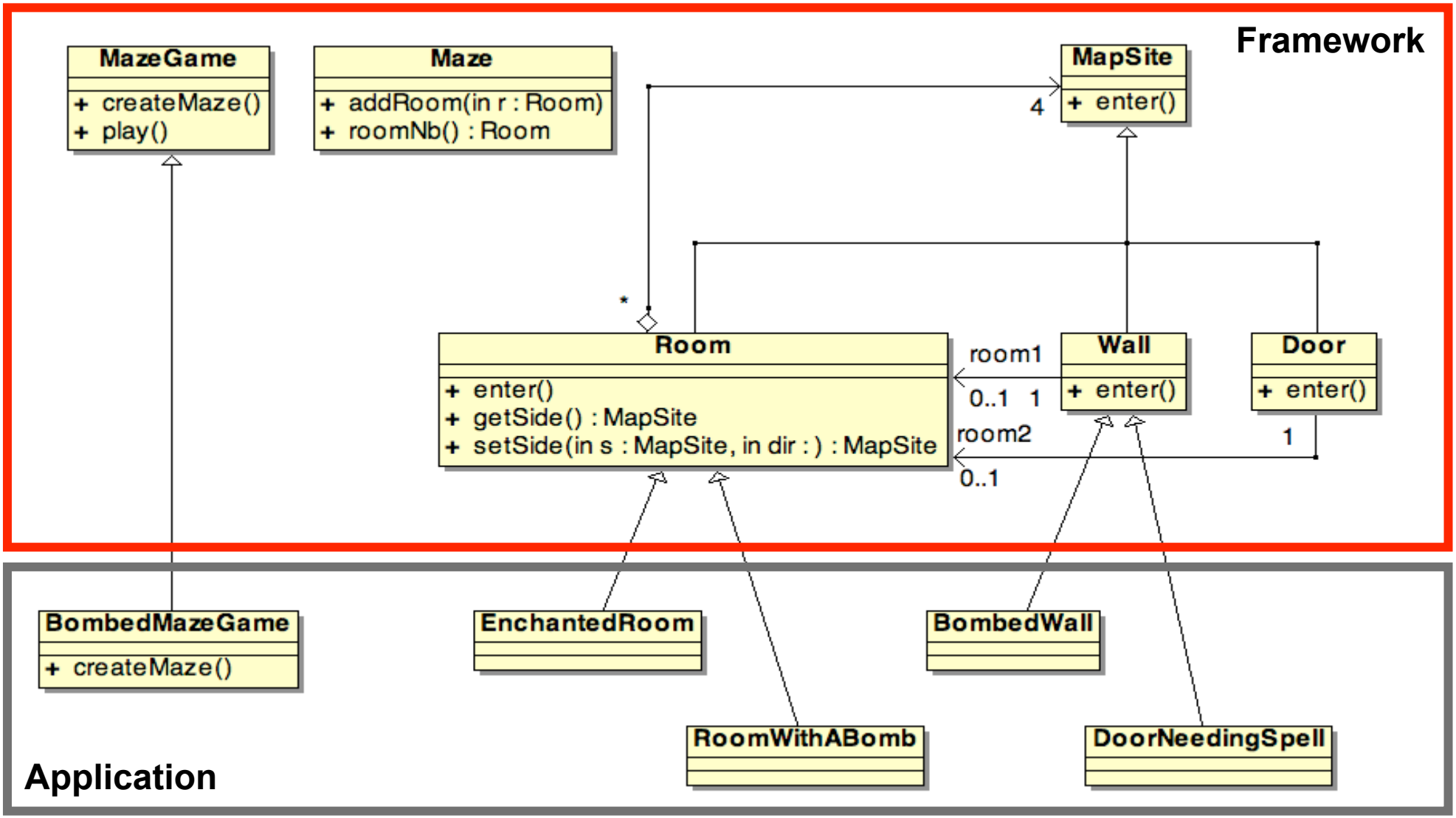
- **Nom du pattern**
  - utilisé pour décrire le pattern, ses solutions et les conséquences en un mot ou deux
- **Problème**
  - description des conditions d 'applications. Explication du problème et de son contexte
- **Solution**
  - description des éléments (objets, relations, responsabilités, collaboration) permettant de concevoir la solution au problème ; utilisation de diagrammes de classes, de séquences, …
    vision statique ET dynamique de la solution
- **Conséquences**
  - description des résultats (effets induits) de l 'application du pattern sur le système (effets positifs ET négatifs)

Design Patterns de création

Design Patterns de création

- Rendre le système indépendant de la manière dont les objets sont créés, composés et représentés
  - Encapsulation de la connaissance des classes concrètes à utiliser
  - Cacher la manière dont les instances sont créées et combinées
- Permettre *dynamiquement* ou *statiquement* de préciser **QUOI** (l'objet), **QUI** (l'acteur), **COMMENT** (la manière) et **QUAND** (le moment) de la création
- Deux types de motifs
  1. Motifs de création de classe (utilisation de l'héritage) : Factory
  2. Motifs de création d'objets (délégation de la construction à un autre objet) : AbstractFactory, Builder, Prototype

Exemple : Labyrinthe



Exemple : Labyrinthe

```
class MazeGame {
   void Play() {…}

   public Maze createMaze() {
      Maze aMaze = new Maze();
      Room r1 = new Room(1);
      Room r2 = new Room(2);
      Door theDoor = new Door(r1, r2);
      aMaze.addRoom(r1);
      aMaze.addRoom(r2);
      r1.setSide(North, new Wall());
      r1.setSide(East, theDoor);
      r1.setSide(South, new Wall());
      r1.setSide(West, new Wall());
      r2.setSide(North, new Wall());
      r2.setSide(East, new Wall());
      r2.setSide(South, new Wall());
      r2.setSide(West, theDoor);
      return aMaze;
   }
}
```

```
class BombedMazeGame extends MazeGame {
…
   public Maze createMaze() {
      Maze aMaze = new Maze();
      Room r1 = new RoomWithABomb(1);
      Room r2 = new RoomWithABomb(2);
      Door theDoor = new Door(r1, r2);
      aMaze.addRoom(r1);
      aMaze.addRoom(r2);
      r1.setSide(North, new BombedWall());
      r1.setSide(East, theDoor);
      r1.setSide(South, new BombedWall());
      r1.setSide(West, new BombedWall());
      r2.setSide(North, new BombedWall());
      r2.setSide(East, new BombedWall());
      r2.setSide(South, new BombedWall());
      r2.setSide(West, theDoor);
      return aMaze;
   }
}
```

Factory Method (1)

- **Problème**
  - ce motif est à utiliser dans les situations où existe le besoin de standardiser le modèle architectural pour un ensemble d'applications, tout en permettant à des applications individuelles de définir elles-mêmes leurs propres objets à créer
- **Conséquences**
  - \+ Elimination du besoin de code spécifique à l'application dans le code du framework (uniquement l'interface du Product)
  - – Multiplication du nombre de classes

Factory Method (2)

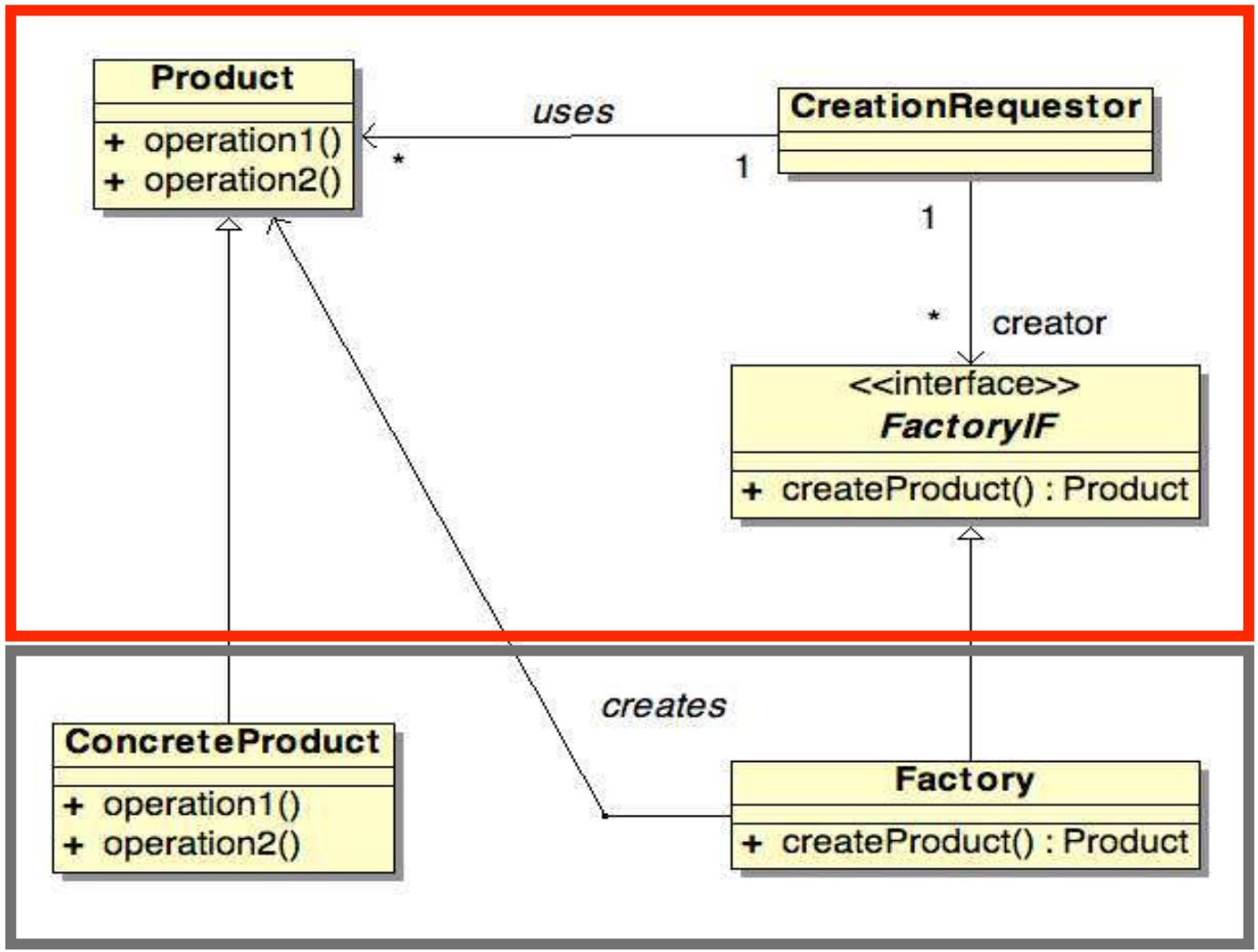
Solution

Product – defines the interface of objects the factory method creates

ConcreteProduct – implements Product interface

FactoryIF – declares the factory method which returns the object of type Product

Factory – overrides the factory method to return an instance of a ConcreteProduct



Exemple : Labyrinthe

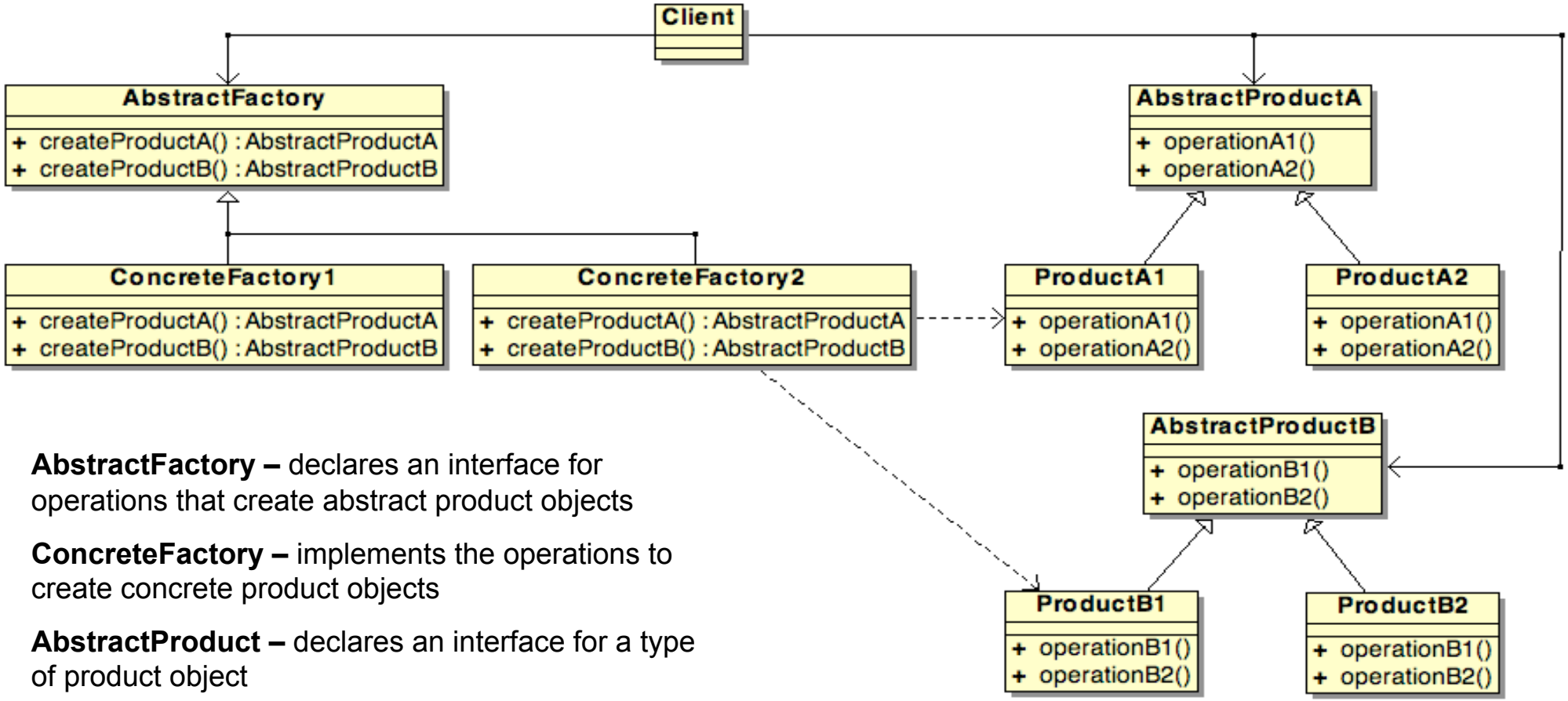
```
class MazeGame {
   void Play() {…}

   public Maze createMaze() {
      Maze aMaze = makeMaze();
      Room r1 = makeRoom(1);
      Room r2 = makeRoom(2);
      Door theDoor = makeDoor(r1, r2);
      aMaze.addRoom(r1);
      aMaze.addRoom(r2);
      r1.setSide(North, makeWall());
      r1.setSide(East, theDoor);
      r1.setSide(South, makeWall());
      r1.setSide(West, makeWall());
      r2.setSide(North, makeWall());
      r2.setSide(East, makeWall());
      r2.setSide(South, makeWall());
      r2.setSide(West, theDoor);
      return aMaze;
   }
}
```

```
class BombedMazeGame extends MazeGame {
…
   public Wall makeWall() {
      return new BombedWall();
   }

   public Room makeRoom(int i) {
      return new RoomWithABomb(i);
   }
}
```

Abstract Factory (1)

- **Problème**
  - ce motif est à utiliser dans les situations où existe le besoin de travailler avec des familles de produits tout en étant indépendant du type de ces produits
  - doit être configuré par une ou plusieurs familles de produits
- **Conséquences**
  - \+ Séparation des classes concrètes, des classes clients
    - les noms des classes produits n'apparaissent pas dans le code client
    - Facilite l'échange de familles de produits
    - Favorise la cohérence entre les produits
  - \+ Le processus de création est clairement isolé dans une classe
  - – la mise en place de nouveaux produits dans l'AbstractFactory n'est pas aisée
- **Exemple**
  - java.awt.Toolkit

Abstract Factory (2)



AbstractFactory – declares an interface for operations that create abstract product objects

ConcreteFactory – implements the operations to create concrete product objects

AbstractProduct – declares an interface for a type of product object

ConcreteProduct
- **–** defines a product object to be created by the corresponding concrete factory
- **–** implements the AbstractProduct interface

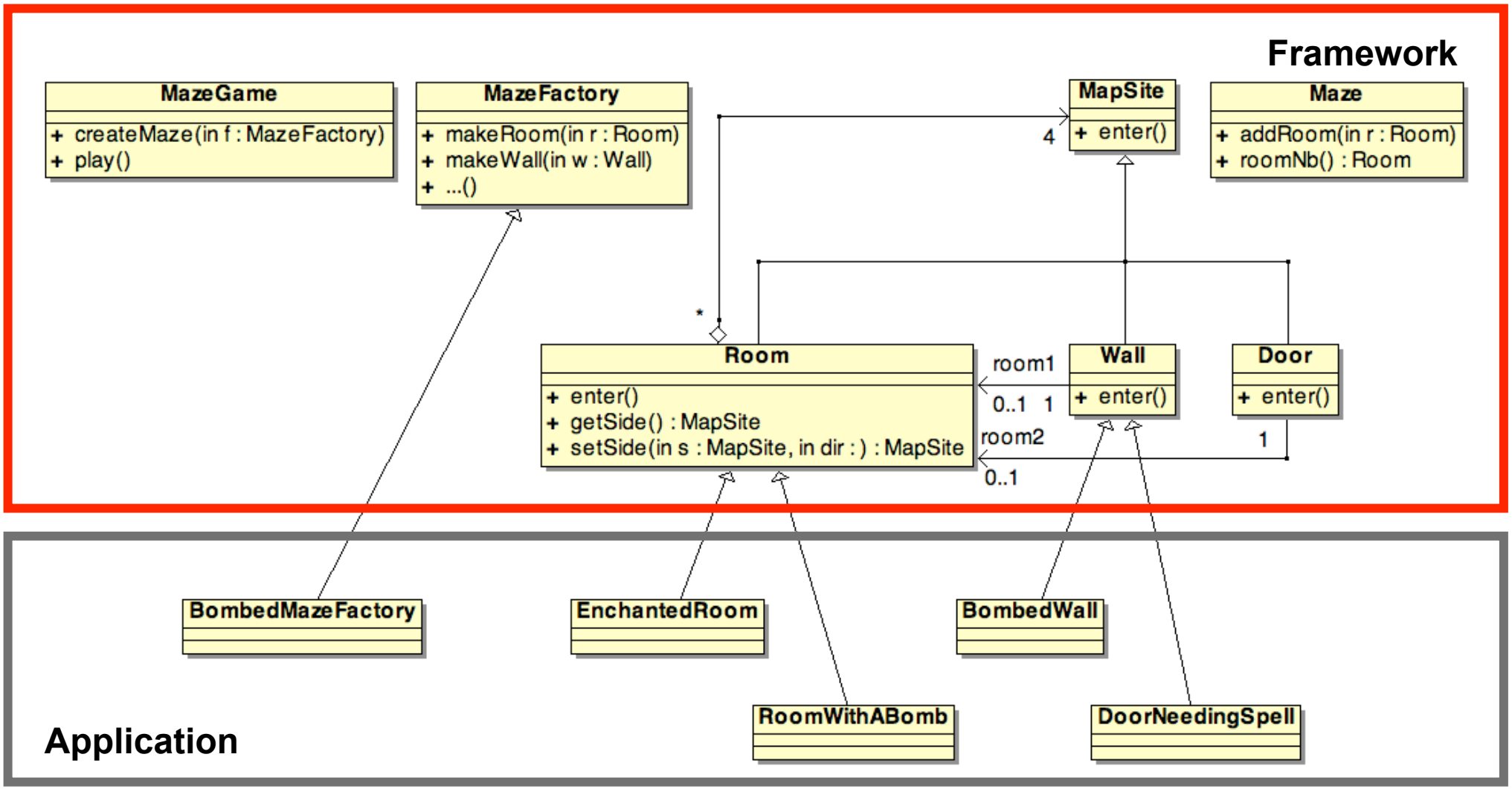
Client – uses interfaces declared by AbstractFactory and AbstractProduct classes

Exemple : Labyrinthe

```
class MazeGame {
   void Play() {…}

   public Maze createMaze(MazeFactory f) {
      Maze aMaze = f.makeMaze();
      Room r1 = f.makeRoom(1);
      Room r2 = f.makeRoom(2);
      Door theDoor = f.makeDoor(r1, r2);
      aMaze.addRoom(r1);
      aMaze.addRoom(r2);
      r1.setSide(North, f.makeWall());
      r1.setSide(East, theDoor);
      r1.setSide(South, makeWall());
      r1.setSide(West, makeWall());
      r2.setSide(North, makeWall());
      r2.setSide(East, makeWall());
      r2.setSide(South, makeWall());
      r2.setSide(West, theDoor);
      return aMaze;
  }
}
```

```
class BombedMazeFactory extends MazeFactory {
…
   public Wall makeWall() {
      return new BombedWall();
   }

   public Room makeRoom(int i) {
      return new RoomWithABomb(i);
   }
}
```

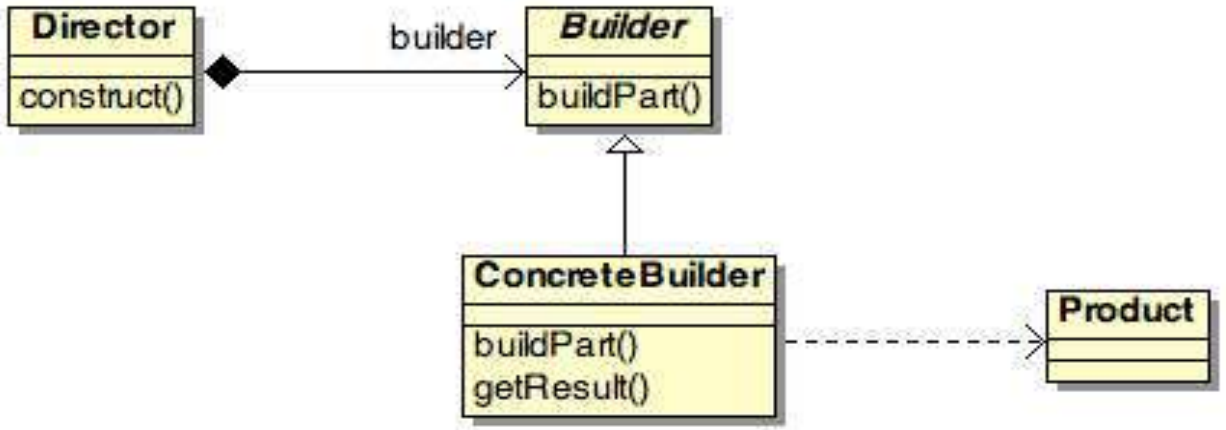
Exemple : Labyrinthe + Factory



Builder (1)

- **Problème**
  - ce motif est intéressant à utiliser lorsque l'algorithme de création d'un objet complexe doit être indépendant des constituants de l'objet et de leurs relations, ou lorsque différentes représentations de l'objet construit doivent être possibles
- **Conséquences**
  - \+ Variation possible de la représentation interne d'un produit
    - l'implémentation des produits et de leurs composants est cachée au Director
    - Ainsi la construction d'un autre objet revient à définir un nouveau Builder
  - \+ Isolation du code de construction et du code de représentation du reste de l'application
  - \+ Meilleur contrôle du processus de construction

Builder (2)



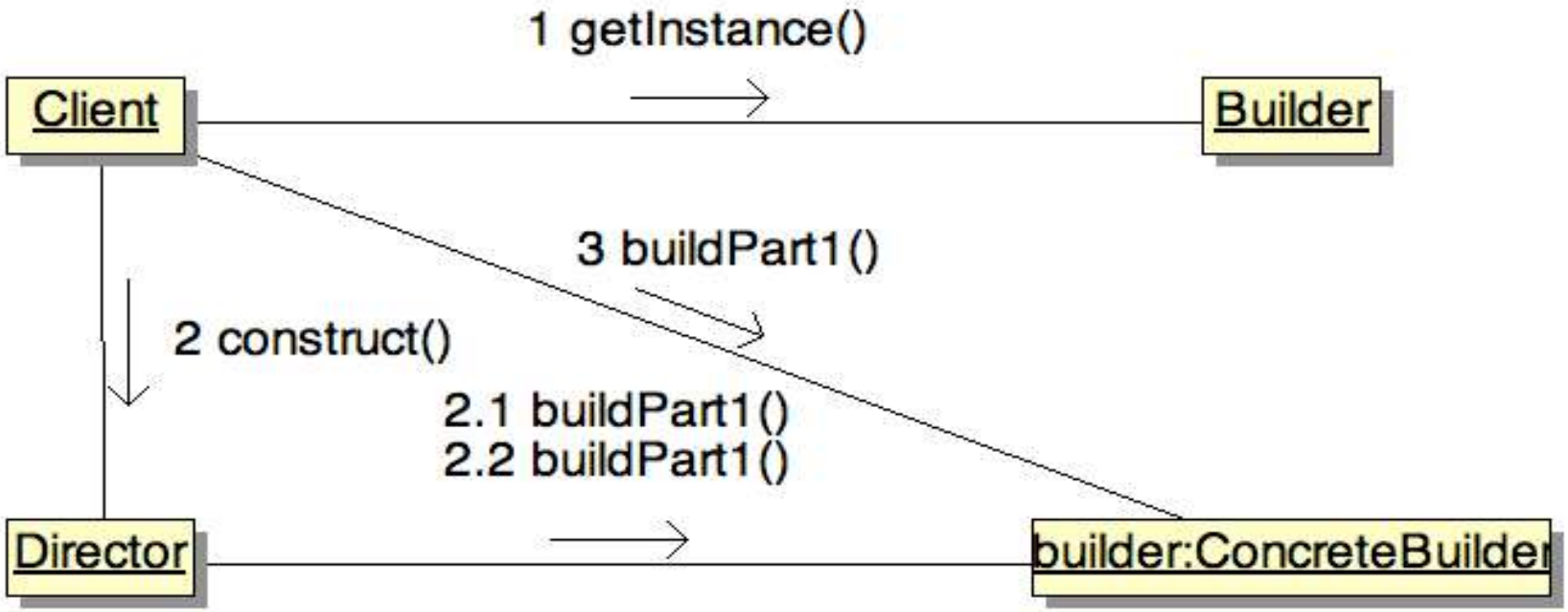
Builder – interface for creating parts of a Product object

ConcreteBuilder – constructs and assembles parts of the product by implementing the Builder interface

Director – constructs an object using Builder Interface

Product – represents the complex object under construction

Builder (3)


1 getInstance()
Client
Builder
3 buildPart1()
2 construct()
2.1 buildPart1()
2.2 buildPart1()
Director
builder:ConcreteBuilder

Exemple : Labyrinthe

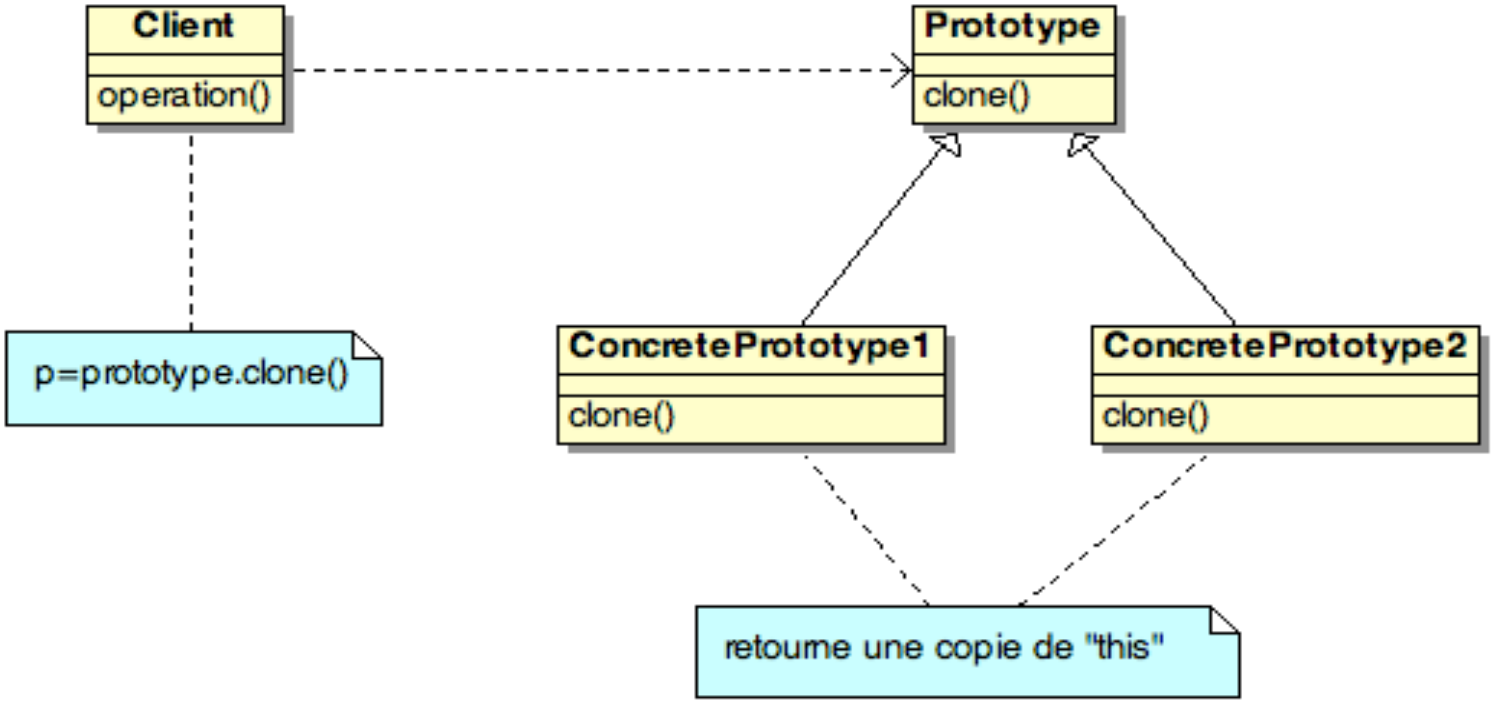
```
class MazeGame {
   void Play() {…}

   public Maze createMaze(MazeBuilder builder) {
      builder.BuildMaze();
       builder.BuildRoom(1);
      builder.BuildRoom(2);
      builder.BuildDoor(1,2);
      return builder.GetMaze();
  }
}
```

```
class MazeBuilder {
   public void BuildMaze () { }
   public void BuildRoom(int room) { }
   public void BuildDoor(int rf, int rt) { }

   public Mase GetMaze() {return null;}
   protected MazeBuilder() {…}
}

class StandardMazeBuilder extends MazeBuilder {
   public void BuildMaze () { … }
   public void BuildRoom(int room) { … }
   public void BuildDoor(int rf, int rt) { … }

   public Mase GetMaze() {return null;}
   public StandardMazeBuilder() {
     _currentMaze = new Maze;
   }

   private Direction CommonWall(Room r1, Room r2) {…}
   private Maze _currentMaze;
}
```

Prototype (1)

- **Problème**
  - Le système doit être indépendant de la manière dont ses produits sont créés, composés et représentés : les classes à instancier sont spécifiées au moment de l'exécution
  - La présence de hiérarchies de Factory similaires aux hiérarchies de produits doivent être évitées. Les combinaisons d'instances sont en nombre limité
- **Conséquences**
  - mêmes conséquences que Factory et Builder
- **Exemple**
  - java.lang.Cloneable

Prototype (2)



Prototype – declares an interface for cloning itself

ConcretePrototype – implements an operation for cloning itself

Client – creates a new object by asking a prototype to clone itself

Exemple : Bilan sur le Labyrinthe

- **Factory** : CreateMaze a un objet en paramètre utilisé pour créer les composants du labyrinthe, on peut changer les composants de la structure du labyrinthe en passant un autre paramètre
- **Builder** : CreateMaze a un objet paramètre capable de créer lui même un labyrinthe dans sa totalité, il est possible de changer la structure du labyrinthe en dérivant un nouvel objet
- **FactoryMethod** : CreateMaze appelle des fonctions virtuelles au lieu de constructeurs pour créer les composants du labyrinthe, il est alors possible de modifier la création en dérivant une nouvelle classe et en redéfinissant ces fonctions virtuelles
- **Prototype** : CreateMaze est paramétré par des composants prototypiques qu'il copie et ajoute au labyrinthe, il est possible de changer la structure du labyrinthe en fournissant d'autres composants

Singleton

- **Problème**
  - avoir une seule instance d'une classe et pouvoir l'accéder et la manipuler facilement
- **Solution**
  - une seule classe est nécessaire pour écrire ce motif
- **Conséquences**
  - l'unicité de l'instance est complètement contrôlée par la classe elle même. Ce motif peut facilement être étendu pour permettre la création d'un nombre donné d'instances

Singleton (2)

```
// Only one object of this class can be created
class Singleton {
   private static Singleton instance = null;

   private Singleton() {
      ...
   }

   public static Singleton getInstance() {
      if (instance == null)
         instance = new Singleton();
      return instance;
   }
   ...
}

class Program {
   public void aMethod() {
      Singleton X = Singleton.getInstance();
   }
}
```

Résumé : DP de création

- Le **Factory Pattern** est utilisé pour choisir et retourner une instance d ’une classe parmi un nombre de classes similaires selon une donnée fournie à la factory
- Le **Abstract Factory Pattern** est utilisé pour retourner un groupe de classes
- Le **Builder Pattern** assemble un nombre d’objets pour construire un nouvel objet, à partir des données qui lui sont présentées. Fréquemment le choix des objets à assembler est réalisé par le biais d’une Factory
- Le **Prototype Pattern** copie ou clone une classe existante plutôt que de créer une nouvelle instance lorsque cette opération est coûteuse
- Le **Singleton Pattern** est un pattern qui assure qu’il n’y a qu’une et une seule instance d’un objet et qu’il est possible d’avoir un accès global à cette instance

Design Patterns de structure

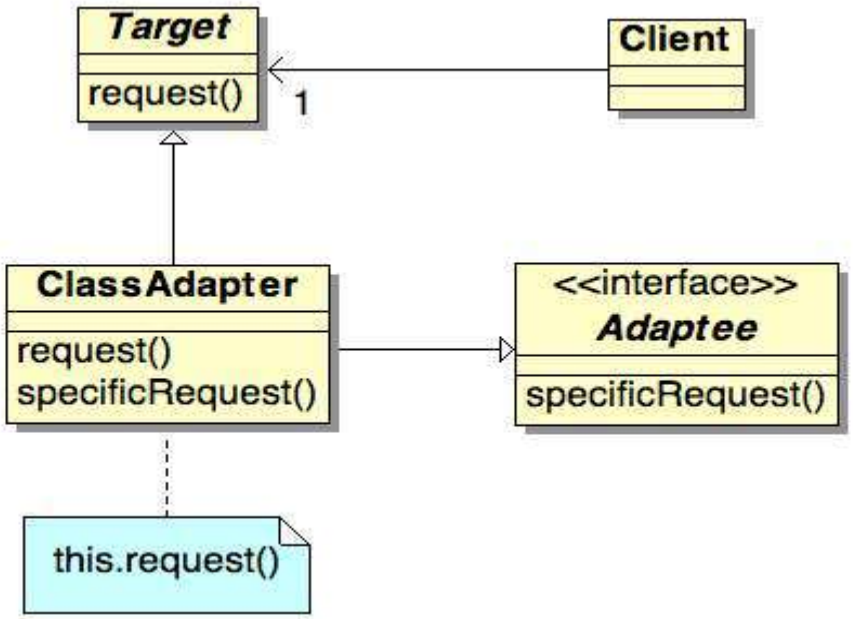
Design Patterns de structure

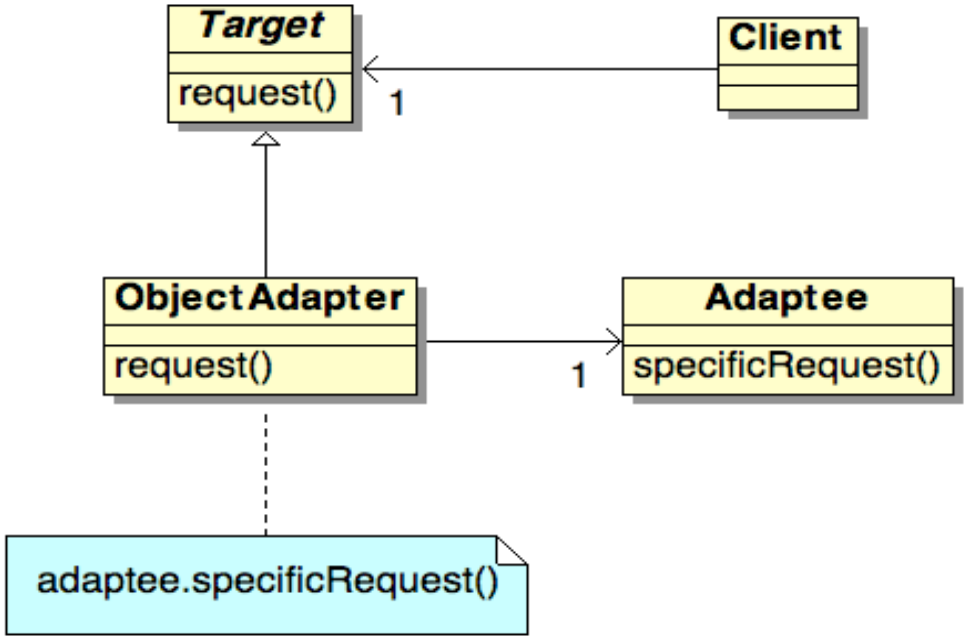
- Abstraction de la manière dont les classes et les objets sont composés pour former des structures plus importantes.
- Deux types de motifs :
  - Motifs de structure de classes
    - Utilisation de l'héritage pour composer des interfaces et/ou des implémentations (ex : Adapter).
  - Motifs de structure d'objets
    - composition d'objets pour réaliser de nouvelles fonctionnalités :
      - ajouter d'un niveau d'indirection pour accéder à un objet
        ex : Adapter d'objet, Bridge, Facade, Proxy,
      - composition récursive pour organiser un nombre quelconque d'objets
        ex : Composite

Adapter (1)

- **Problème**
  - Utilisation d'une classe existante dont l'interface ne nous convient pas (→ convertir l'interface d'une classe en une autre)
  - Utilisation de plusieurs sous-classes dont l'adaptation des interfaces est impossible par dérivation (→ Object Adapter)
- **Conséquences**
  - Adapter de classe
    - \+ il n'introduit qu'une nouvelle classe, une indirection vers la classe adaptée n'est pas nécessaire
    - MAIS il ne fonctionnera pas dans le cas où la classe adaptée est racine d'une dérivation
  - Adapter d'objet
    - \+ il peut fonctionner avec plusieurs classes adaptées
    - MAIS il peut difficilement redéfinir des comportements de la classe adaptée

Adapter (2)





Target – defines the domain-specific interface that client uses.

Client – collaborates with objects conforming to the Target interface.

Adaptee – defines an existing interface that needs adapting.

Adapter – adapts the interface of Adaptee to the Target interface.

Adapter (3)

```
interface Stack {
   void push(Object o);
   Object pop();
   Object top();
}

/* Liste doublement chaînée */
class DList {
   public void insert (DNode pos, Object o) { ... }
   public void remove (DNode pos) { ... }
   public void insertHead (Object o) { ... }
   public void insertTail (Object o) { ... }
   public Object removeHead () { ... }
   public Object removeTail () { ... }
   public Object getHead () { ... }
   public Object getTail () { ... }
}
```

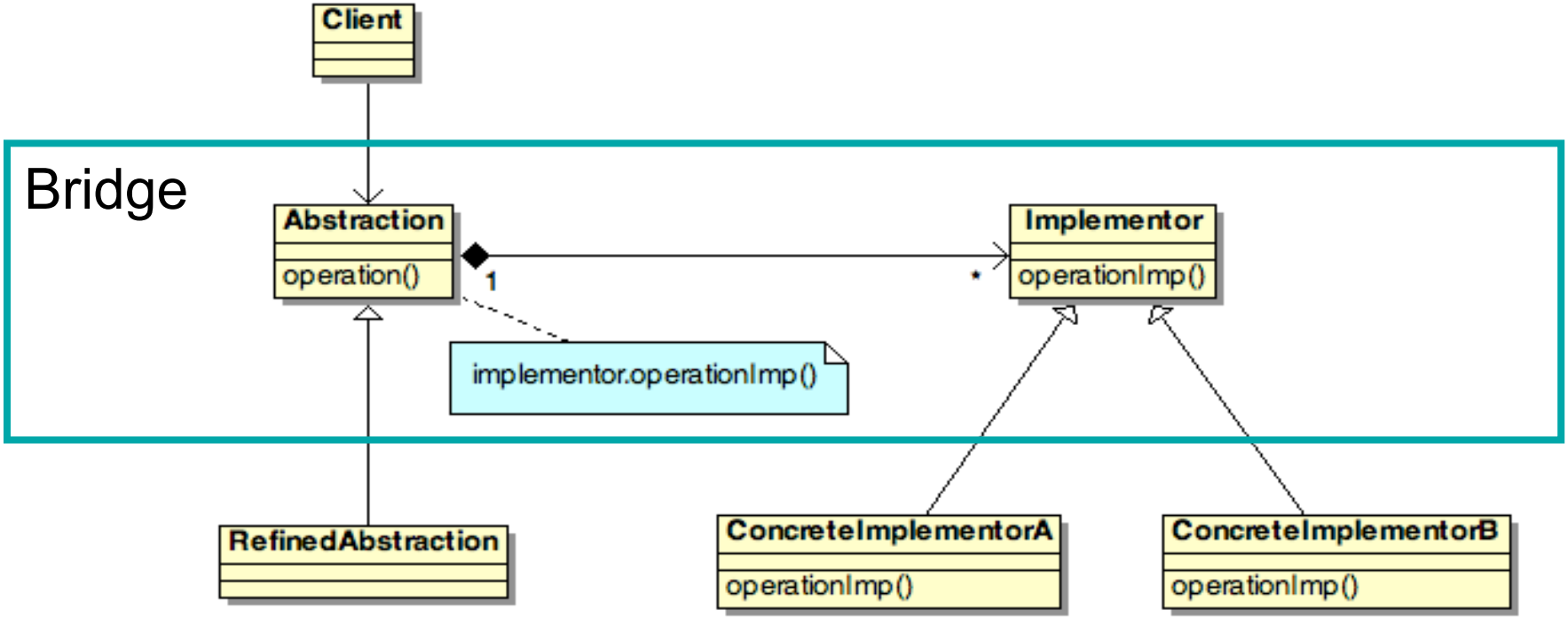
Comment adapter une liste doublement chaînée en une pile ?

```
/* Adapt DList class to Stack interface */
class DListImpStack extends DList implements Stack {
   public void push(Object o) {
      insertTail(o);
   }
   public Object pop() {
      return removeTail();
   }
   public Object top() {
      return getTail();
   }
}
```

Bridge (1)

- **Problème**
  - ce motif est à utiliser lorsque l'on veut découpler l'implémentation de l'abstraction de telle sorte que les deux puissent varier indépendamment
- **Conséquences**
  + interfaces et implémentations peuvent être couplées/ découplées lors de l'exécution

Bridge (2)



Abstraction – defines the abstraction's interface.

RefinedAbstraction – Extends the interface defined by Abstraction.

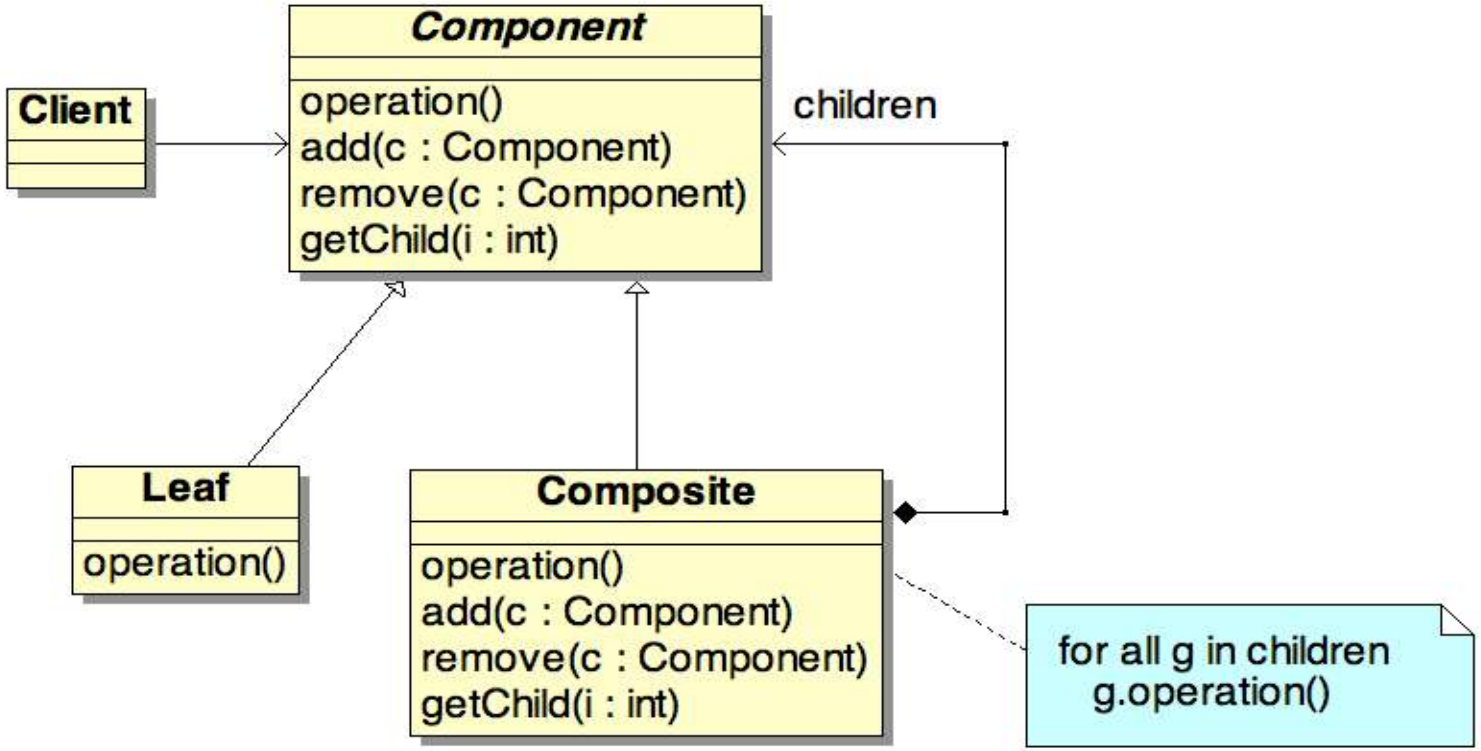
Implementor – defines the interface for implementation classes.

ConcreteImplementor – implements the Implementor interface and defines its concrete implementation.

Composite (1)

- **Problème**
  - établir des structures arborescentes entre des objets et les traiter uniformément
- **Conséquences**
  - \+ hiérarchies de classes dans lesquelles l'ajout de nouveaux composants est simple
  - \+ simplification du client qui n'a pas à se préoccuper de l'objet accédé
  - – MAIS il est difficile de restreindre et de vérifier le type des composants
- **Exemple**
  - – java.awt.Component
  - – java.awt.Container

Composite (2)



Component – declares the interface for objects in the composition

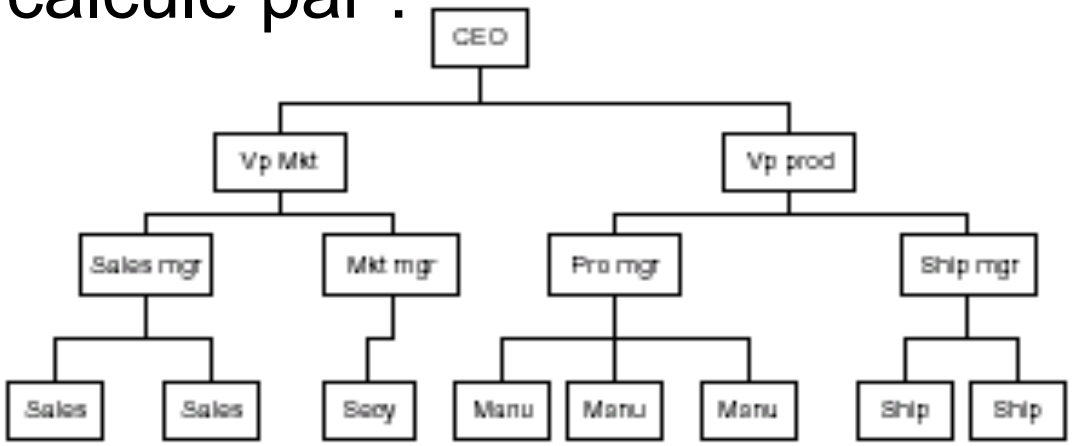
Leaf – represents leaf objects in the composition

Composite – defines behavior for components having children

Client – manipulates objects in the composition through the Component interface

Composite : Exercice

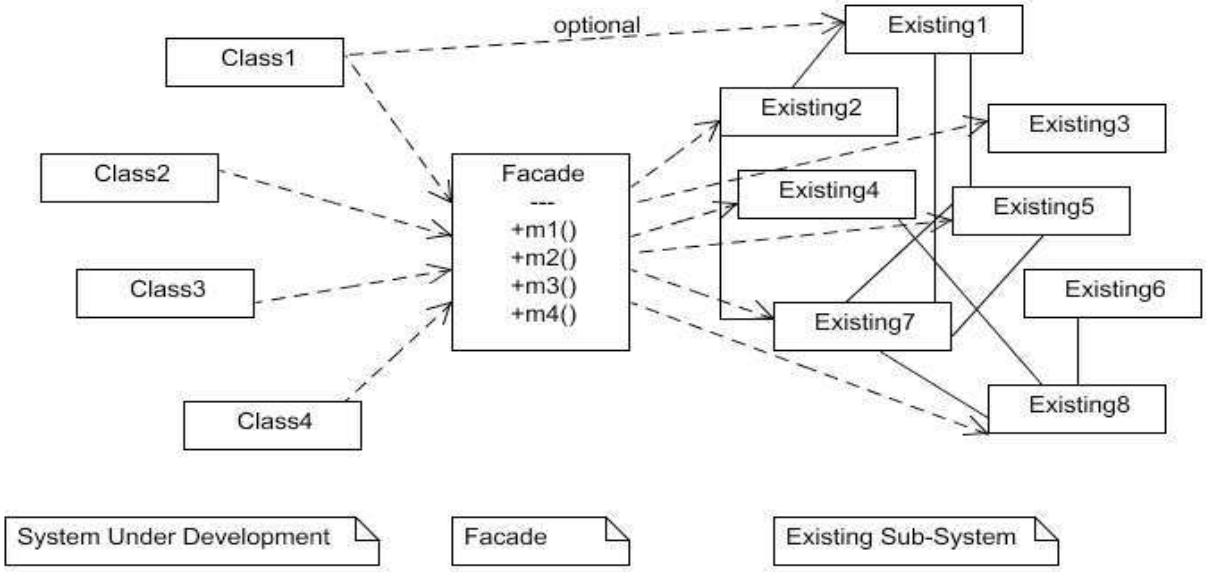
- Chacun des membres de la compagnie reçoit un salaire.
- A tout moment, il doit être possible de demander le coût d'un employé.
- Le coût d'un employé est calculé par :
  - Le coût d'un individu est son salaire.
  - Le coût d'un responsable est son salaire plus celui de ses subordonnés.



Façade (1)

- **Problème**
  - ce motif est à utiliser pour faciliter l'accès à un grand nombre de modules en fournissant une couche interface
- **Conséquences**
  - \+ facilite l'utilisation de sous-systèmes
  - \+ favorise un couplage faible entre des classes et l'application
  - – MAIS des fonctionnalités des classes interfacées peuvent être perdues selon la manière dont est réalisée la Façade

Façade (2)



Facade

– knows which subsystem classes are responsible for a request

– delegates client requests to appropriate subsystem objects

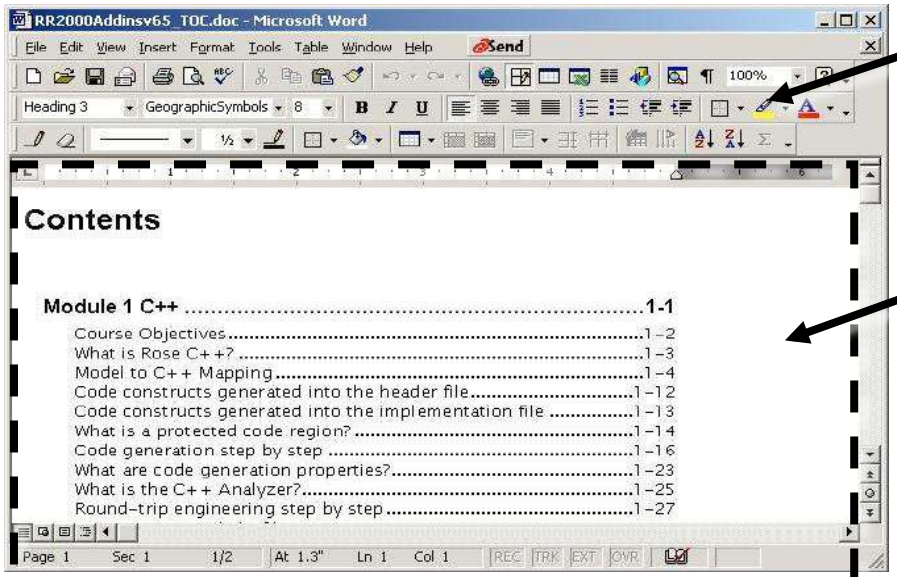
Subsystem classes

– implement subsystem functionality

– handle work assigned by the Façade object

– have no knowledge of the façade; that is, keep no references to it

Proxy (1)



éditeur

Portion visible du document

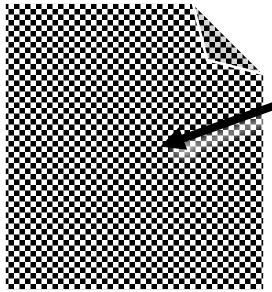
Portion invisible

proxy

image stockée dans une base de donnée

Course Objectives 2-2
What Is Rose CORBA? 2-4
IncludePath 2-5
Project Properties 2-6
Assign class to component 2-7
Where is the code placed? 2-8
Generate the code 2-9
What is reverse engineering? 2-12

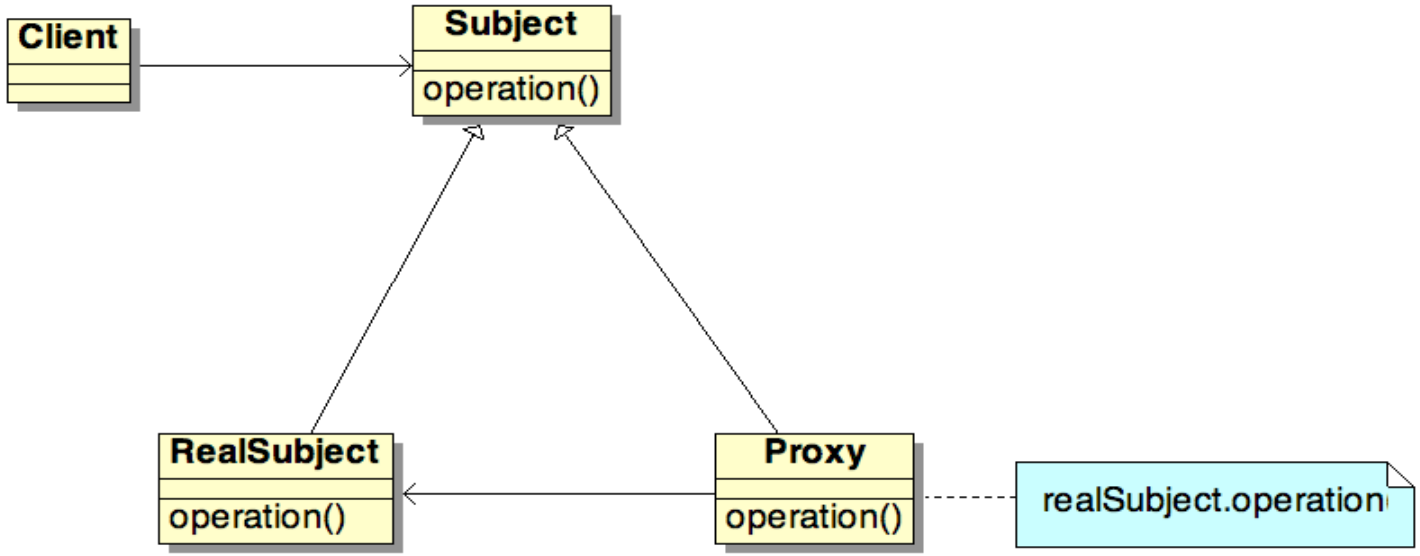
Using Rose CORBA help files
Course Objectives
What is Rose 2000 J?
Class Path Tab
Detail Tab
Style Tab
Assign class to component
Generate the code
What is reverse engineering?



Proxy (2)

- **Problème**
  - ce motif est à utiliser pour agir par procuration pour un objet afin de contrôler les opérations qui lui sont appliquées
    - Masquer des problèmes d'accès (ex : fichier)
    - Différer l'exécution d'opérations coûteuses
    - Contrôler les droits d'accès
- **Conséquences**
  + ajout d'un niveau d'indirection lors de l'accès d'un objet permettant de cacher le fait que l'objet est dans un autre espace d'adressage, n'est pas créé, …

Proxy (3)



Proxy – maintains a reference that lets the proxy access the real subject. Depending upon the kind of proxy:

Remote proxies – implements the Flyweight interface and adds storage for intrinsic state, if any

Virtual proxies – may cache additional information about the real subject

Protection proxies – check that the caller has the access permission required to perform the request

Subject – defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected

RealSubject – defines the real object that the proxy represents

Design Patterns de comportement

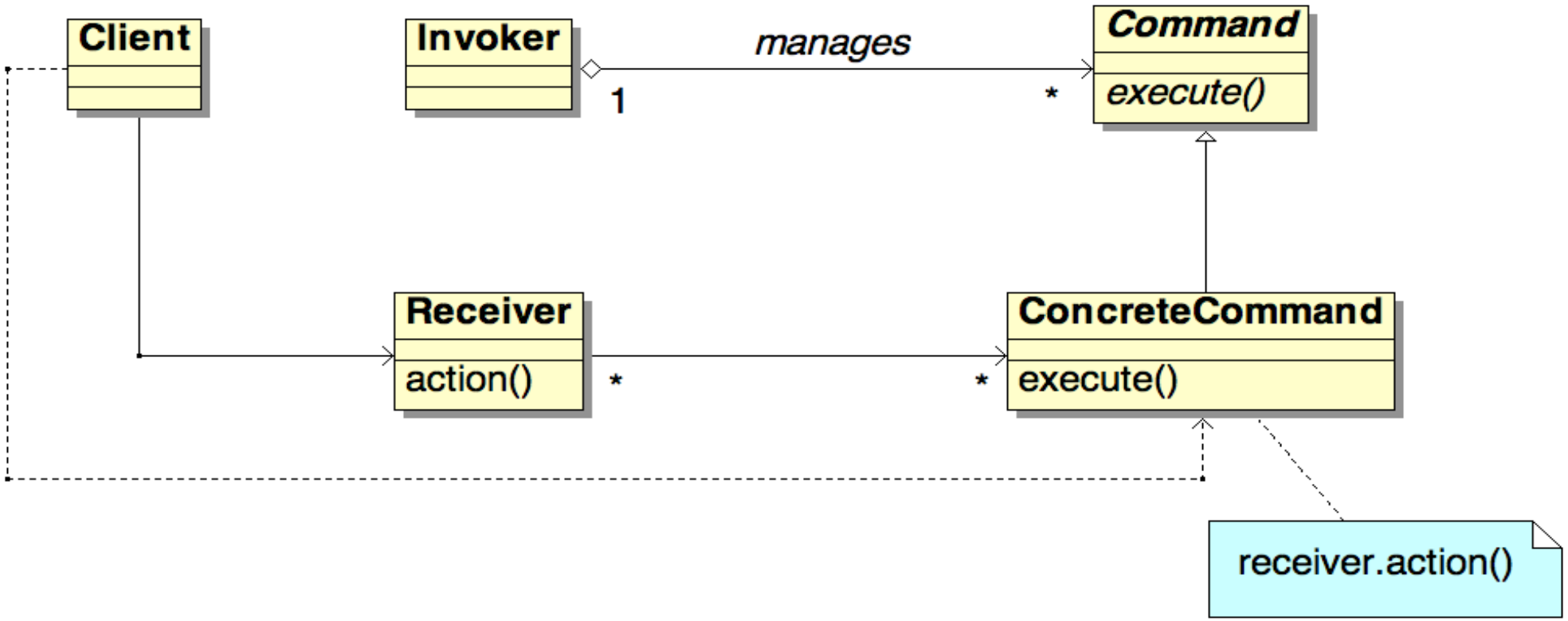
Design Patterns de comportement

- Description de structures d'objets ou de classes avec leurs interactions
- Deux types de motifs
  - Motifs de comportement de classes :
    - utilisation de l'héritage pour répartir les comportements entre des classes (ex : Interpreter)
  - Motifs de comportement d'objets avec l'utilisation de l'association entre objets :
    - pour décrire comment des groupes d'objets coopèrent (ex : Mediator)
    - pour définir et maintenir des dépendances entre objets (ex : Observer)
    - pour encapsuler un comportement dans un objet et déléguer les requêtes à d'autres objets (ex : Strategy, State, Command)
    - pour parcourir des structures en appliquant des comportements (ex : Visitor, Iterator)

Command (1)

- **Problème**
  - on veut effectuer des requêtes sur des objets sans avoir à connaître leur structure
- **Conséquences**
  - \+ découplage entre l'objet qui appelle et celui qui exécute
  - \+ l'ajout de nouvelles commandes est aisée dans la mesure où la modification de classes existantes n'est pas nécessaire

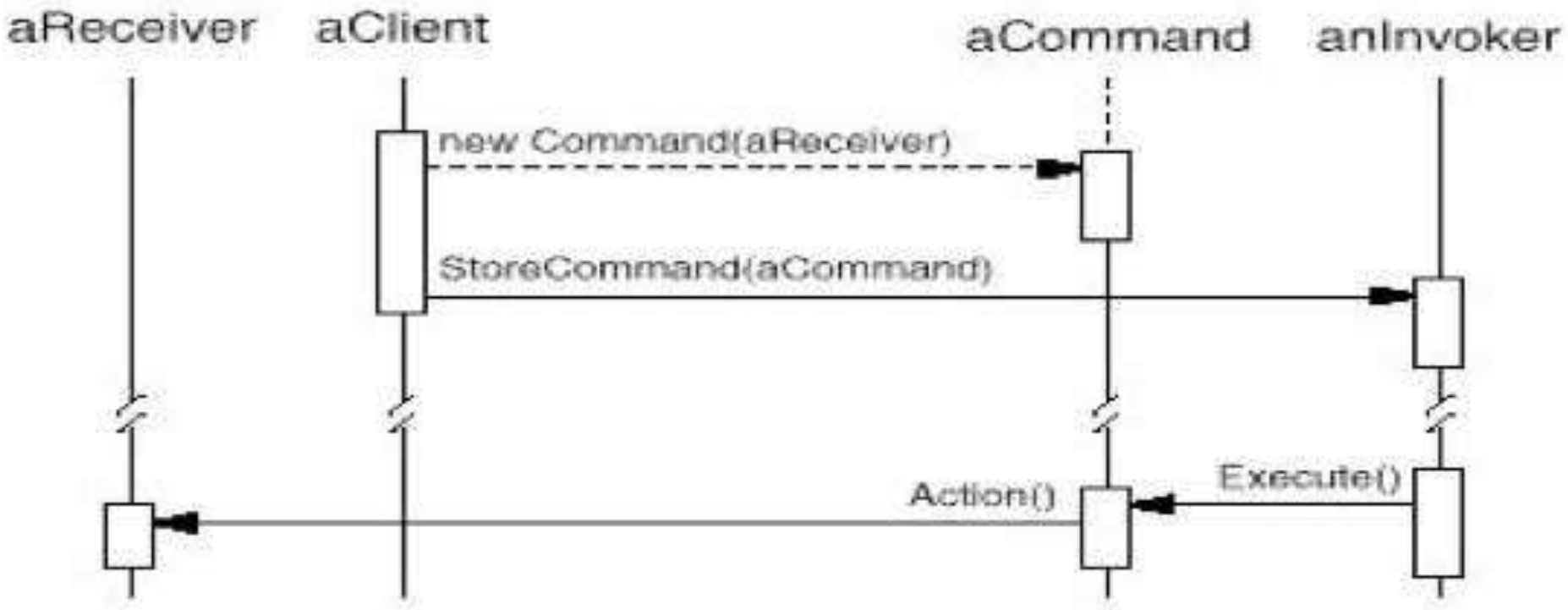
Command (2)



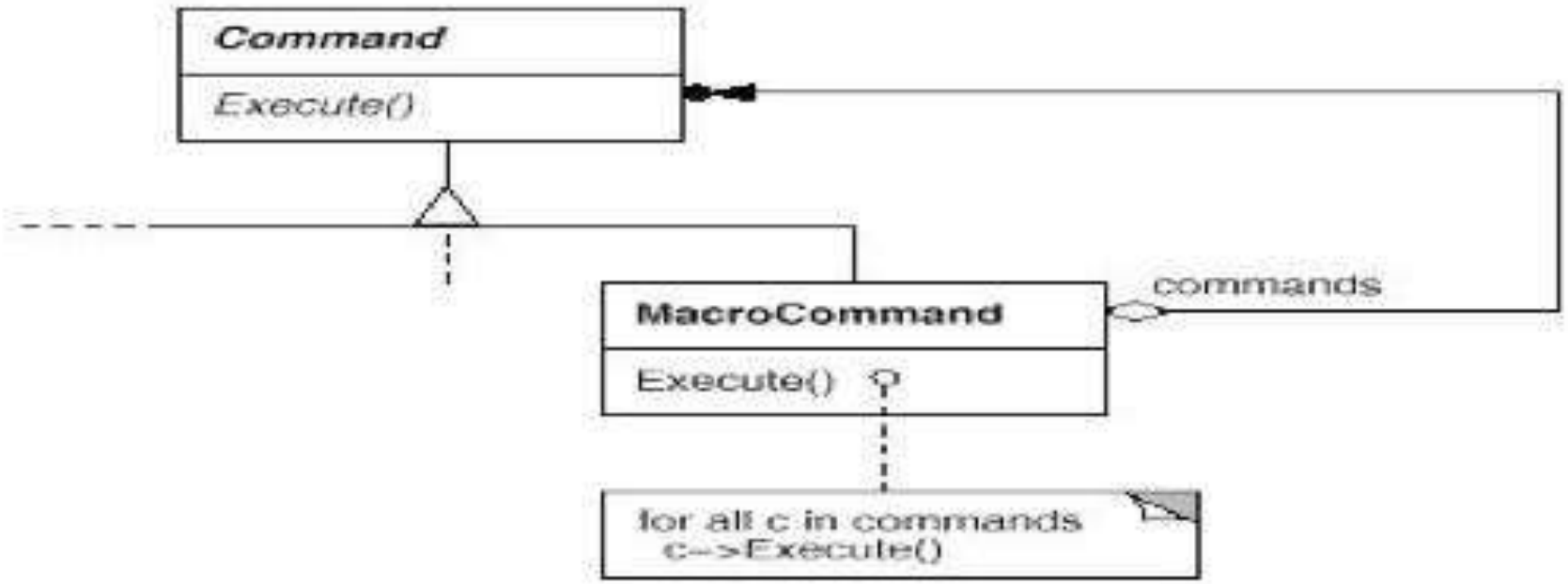
Command – declares an interface for executing an operation
ConcreteCommand – defines a binding between a Receiver object and an action
Client – creates a ConcreteCommand object and sets its receiver

Invoker – asks the command to carry out the request
Receiver – knows how to perform the operations associated with carrying out a request. Any class may server a Receiver

Command (3)


aReceiver
aClient
aCommand
anInvoker
new Command(aReceiver)
StoreCommand(aCommand)
Action()
Execute()

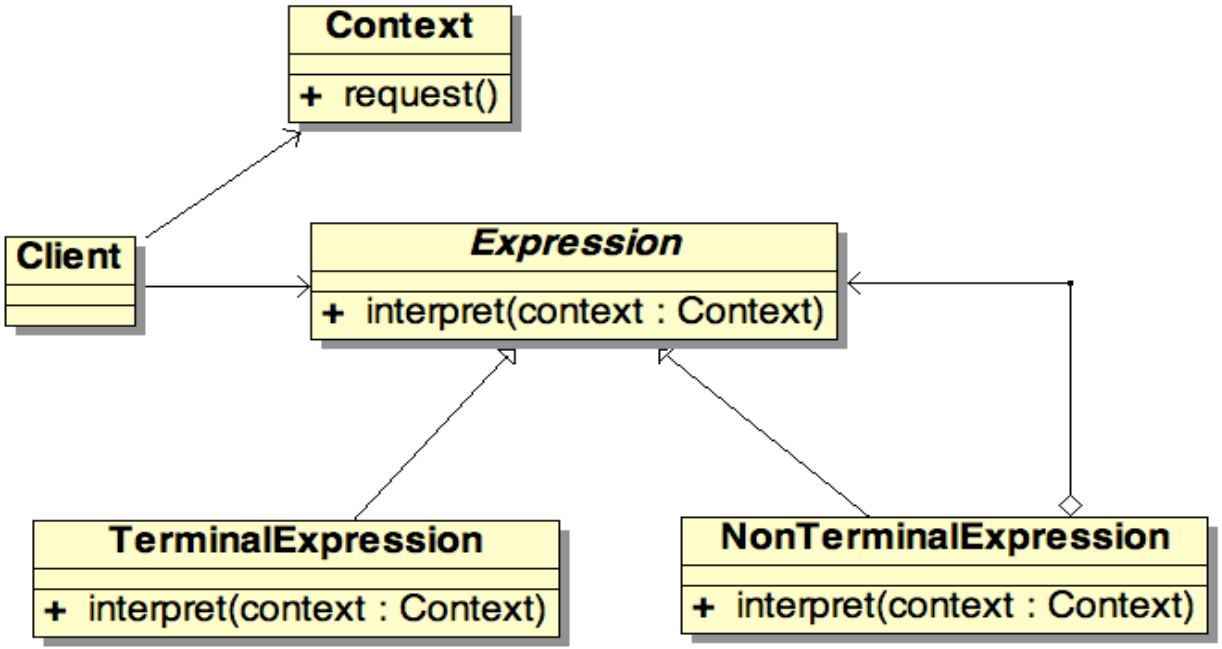
Macro-Command : Command + Composite



Interpreter (1)

- **Problème**
  - ce motif est à utiliser lorsque l'on veut représenter la grammaire d'un langage et l'interpréter, lorsque :
    - La grammaire est simple
    - l'efficacité n'est pas critique
- **Conséquences**
  - \+ facilité de changer et d'étendre la grammaire
  - \+ l'implémentation de la grammaire est simple
  - – MAIS les grammaires complexes sont dures à tenir à jour

Interpreter (2)



AbstractExpression – declares an abstract Interpret operation that is common to all nodes in the abstract syntax tree

TerminalExpression – implements an Interpret operation associated with terminal symbols in the grammar.

NonTerminalExpression – implements an Interpret operation for nonterminal symbols in the grammar

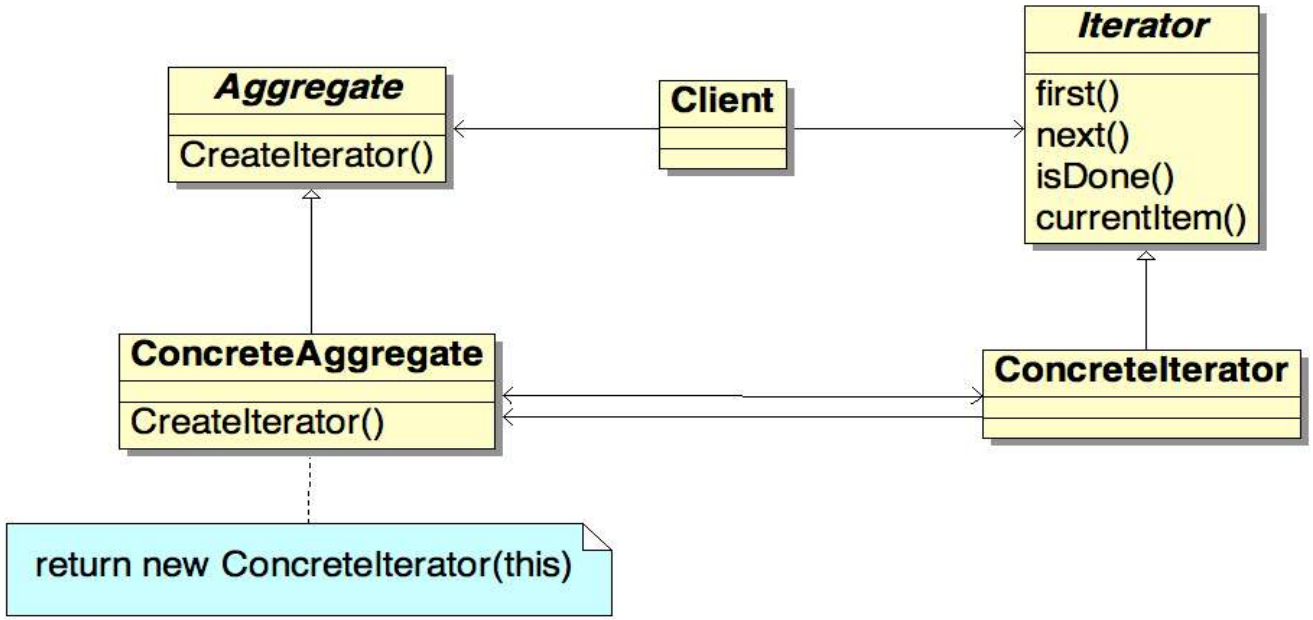
Context – contains info that's global to the interpreter

Client – invokes the Interpret operation

Iterator (1)

- **Problème**
  - ce motif est à utiliser pour parcourir une collection d'éléments sans accéder à sa structure interne
- **Conséquences**
  + des variations dans le parcours d'une collection sont possibles,
  + simplification de l'interface de la collection
  + plusieurs parcours simultanés de la collection sont possibles

Iterator (2)



Iterator
– defines an interface for accessing and traversing elements

ConcreteIterator
– implements the Iterator interface
– keeps track of the current position in the traversal of the aggregate

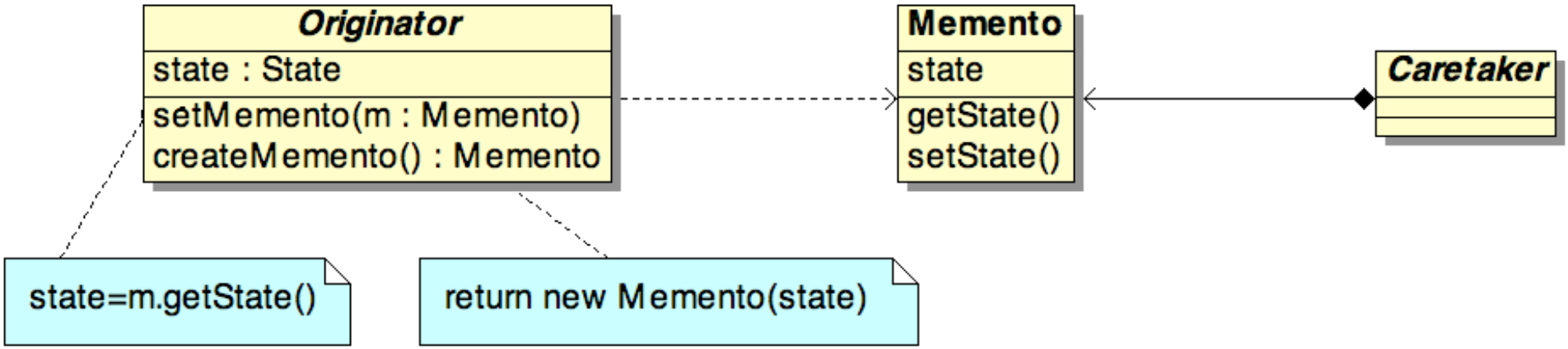
Aggregate
– defines an interface for creating an Iterator object

ConcreteAggregate
– implements the Iterator creation interface to return an instance of the proper ConcreteIterator

Memento (1)

- **Problème**
  - on veut sauvegarder l'état ou une partie de l'état d'un objet
  - **sans** violer le principe d'encapsulation
- **Conséquences**
  + garder les limites de l'encapsulation
  – peut-être coûteux en mémoire et en exécution

Memento (2)



Memento
– stores internal state of the Originator object
– protects against access by objects other than the originator
Originator – creates a memento containing a snapshot of its current internal state

Caretaker
– is responsible for the memento's safekeeping
– never operates on or examines the contents of a memento

Memento (3)

```
class Originator {
    private String state;
    /* lots of memory consumptive private data that is not necessary to define the state and should thus not be
saved. Hence the small memento object. */

    public void set(String state) {
        System.out.println("Originator: Setting state to "+state);
        this.state = state;
    }

    public Object saveToMemento() {
        System.out.println("Originator: Saving to Memento.");
        return new Memento(state);
    }
    public void restoreFromMemento(Object m) {
        if (m instanceof Memento) {
            Memento memento = (Memento)m;
            state = memento.getSavedState();
            System.out.println("Originator: State after restoring from Memento: "+state);
        }
    }

    private static class Memento {
        private String state;

         public Memento(String stateToSave) { state = stateToSave; }
        public String getSavedState() { return state; }
    }
}
```

Memento (4)

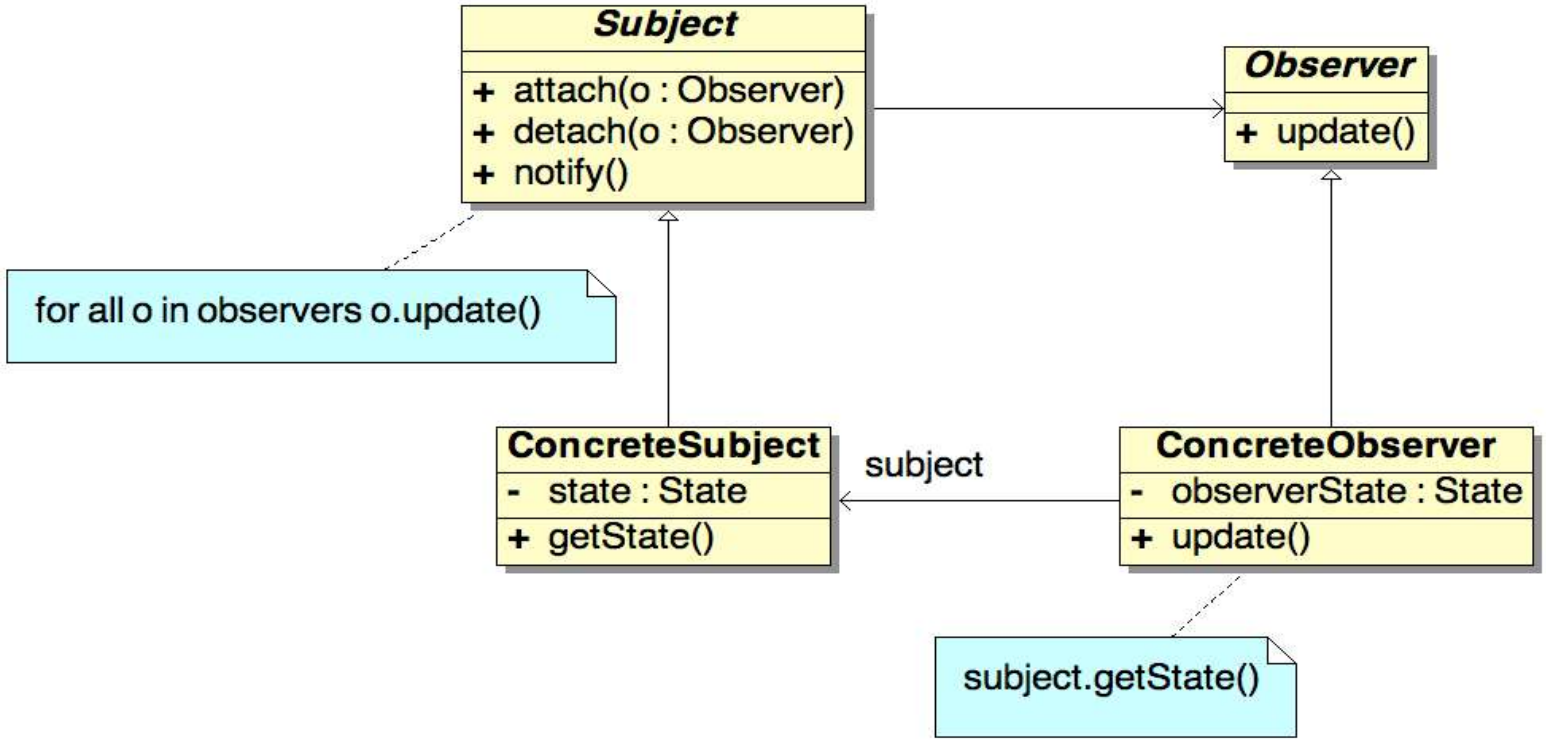
```
class Caretaker {
    private List<Object> savedStates = new ArrayList<Object>();

    public void addMemento(Object m) { savedStates.add(m); }
    public Object getMemento(int index) { return savedStates.get(index); }
}

class MementoExample {
    public static void main(String[] args) {
        Caretaker caretaker = new Caretaker();

        Originator originator = new Originator();
        originator.set("State1");
        originator.set("State2");
        caretaker.addMemento( originator.saveToMemento() );
        originator.set("State3");
        caretaker.addMemento( originator.saveToMemento() );
        originator.set("State4");

        originator.restoreFromMemento( caretaker.getMemento(1) );
    }
}
```

```
Originator: Setting state to State1
Originator: Setting state to State2
Originator: Saving to Memento.
Originator: Setting state to State3
Originator: Saving to Memento.
Originator: Setting state to State4
Originator: State after restoring from Memento: State3
```

Observer (1)

- **Problème**
  - on veut assurer la cohérence entre des classes coopérant entre elles tout en maintenant leur indépendance
- **Conséquences**
  - \+ couplage abstrait entre un sujet et un observeur, support pour la communication par diffusion,
  - – MAIS des mises à jour inattendues peuvent survenir, avec des coûts importants.
- **Exemple**
  - – java.util.Observable
  - – java.util.Observer

Observer (2)



Subject – knows its observers
– provides an interface for attaching and detaching Observer objects
Observer – defines an updating interface for objects that should be notified of changes in a subject

ConcreteSubject – stores state of interest to ConcreteObserver objects
– sends a notification to its observers when its state changes
ConcreteObserver – maintains a reference to a ConcreteSubject object

Observer (3)

```
import java.util.Observable;
...

public class EventSource extends Observable implements Runnable {
   public void run() {
      try {
         final InputStreamReader isr = new InputStreamReader(System.in);
         final BufferedReader br = new BufferedReader(isr);
         while (true) {
            final String response = br.readLine();
            setChanged();
            notifyObservers(response);
          }
       } catch (IOException e) {
          e.printStackTrace();
       }
   }
}
```

```
import java.util.Observable;
import java.util.Observer;

public class ResponseHandler implements Observer {
   private String resp;

   public void update(Observable obj, Object arg) {
      if (arg instanceof String) {
         resp = (String) arg;
         System.out.println("\nReceived Response: "
            + resp);
}
```

```
public class myapp {
   public static void main(String args[]) {
      System.out.println("Enter Text >");
      // création d'un émetteur - lit dans stdin
      EventSource evSrc = new EventSource();
      // création d'un observer
      ResponseHandler respHandler = new ResponseHandler();
      // inscrire l'observeur chez l'émetteur
      evSrc.addObserver(respHandler);
      // exécution du programme
      evSrc.run();
   }
}
```

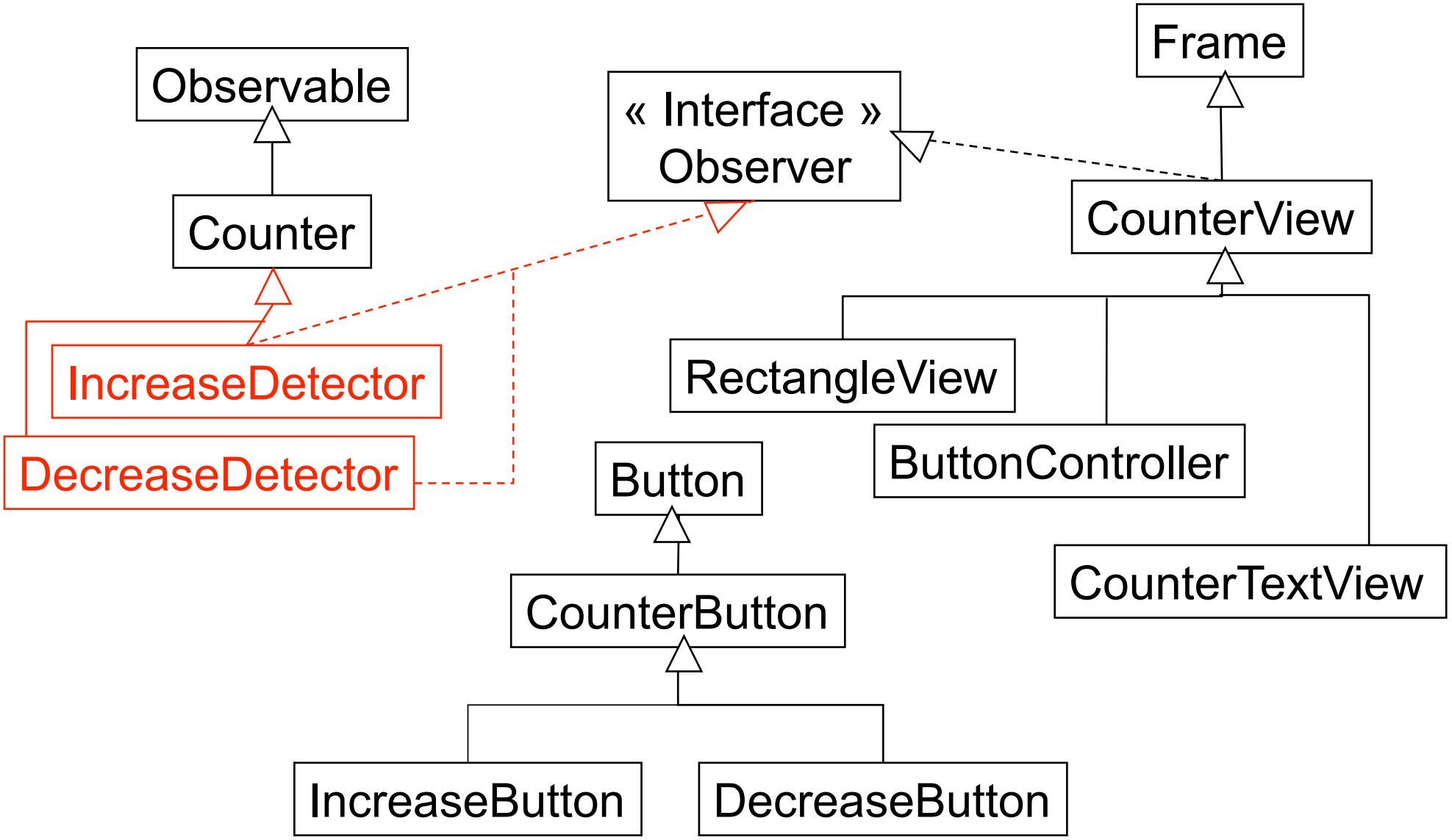
Observer : Exercice (1)

- **Counter** : peut croître ou décroître d'une unité. A chaque fois que le Counter change de valeur, il notifie ses observers du changement.
- **CounterButton** : classe abstraite, permettant de changer la valeur d'un Counter
- **IncreaseButton** (resp. **DecreaseButton**) fait croître (resp. décroître) la valeur de Counter auquel il est attaché à chaque fois que pressé
- **CounterView** : vue observant un Counter. Utilisée comme Parent pour d'autres vues.
- **CounterTextView** : affichage de la valeur d'un compteur en Ascii (sous classe de CounterView)

Observer : Exercice (2)

- **ButtonControler** : fenêtre pour changer la valeur d'un Counter en utilisant les boutons (sous classe de CounterView)
- **RectangleView** : attaché à deux Counters (un pour la largeur et un autre pour la hauteur) (sous classe de CounterView)
- Développer:
  - **IncreaseDetector** (resp. **DecreaseDetector**) : affecté à un ou plusieurs Counter, compte le nombre de fois que leurs valeurs croît (resp. décroît). Il notifie à ses observers des changements.

Observer : Exercice (3)



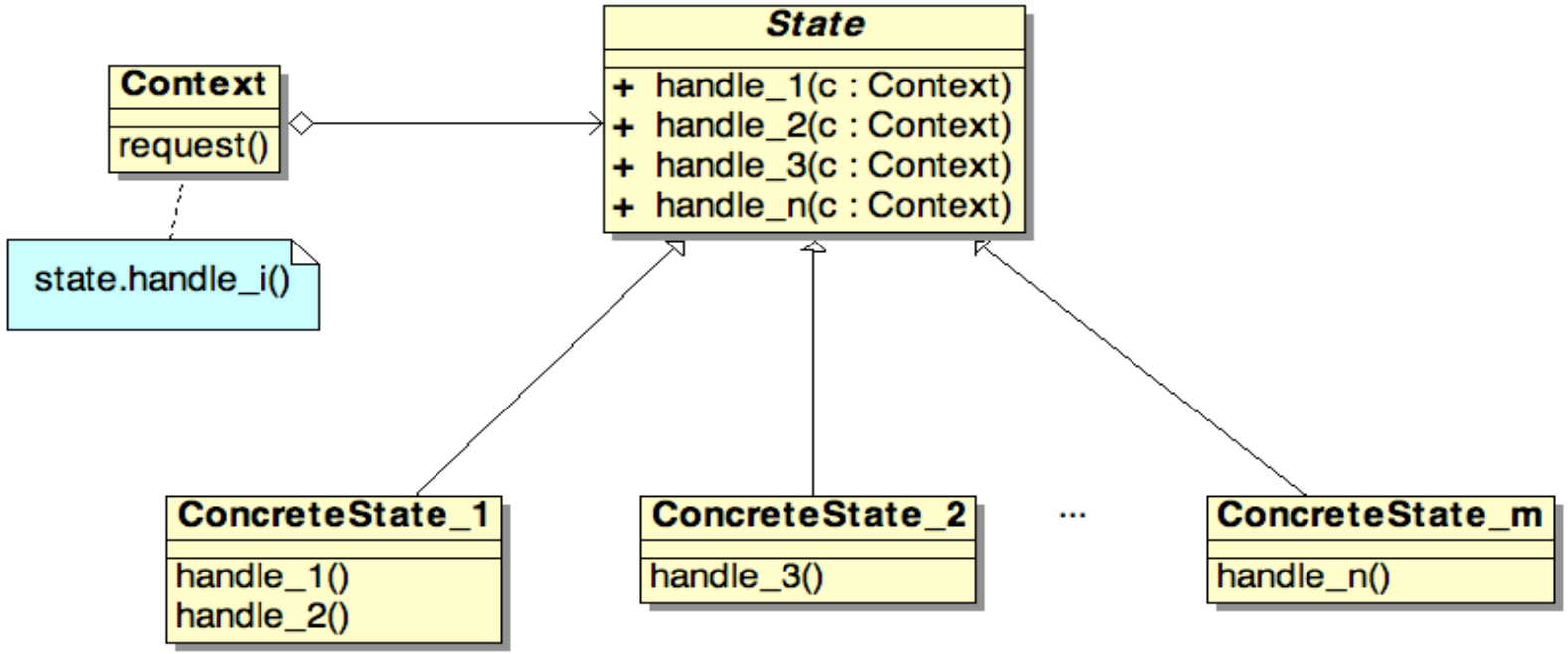
Observer : Exercice (4)

```
Counter x = new Counter( "x" );
Counter y = new Counter( "y" );
IncreaseDetector plus = new IncreaseDetector( "Plus" );
DecreaseDetector moins = new DecreaseDetector( "Moins" )
x.addObserver( plus );
x.addObserver( moins );
y.addObserver( plus );
y.addObserver( moins );
new ButtonController( x,  30, 30, 150, 50 );
new ButtonController( y,  30, 100, 150, 50 );
new CounterTextView( plus, "CounterTextView # of increases",
                           30, 170, 150, 50);
new CounterTextView( moins, "CounterTextView # of decreases",
                           30, 170, 150, 50);
new RectangleView( x, y, 340, 30 );
```

State (1)

- **Problème**
  - ce motif est à utiliser lorsque l'on veut qu'un objet change de comportement lorsque son état interne change
- **Conséquences**
  - \+ possibilité d'ajouter ou de retirer des états et des transitions de manière simple
  - \+ suppression de traitements conditionnels
  - \+ les transitions entre états sont rendues explicites

State (2)



Context
– defines the interface of interest to clients
– maintains an instance of a ConcreteState subclass that defines the current state

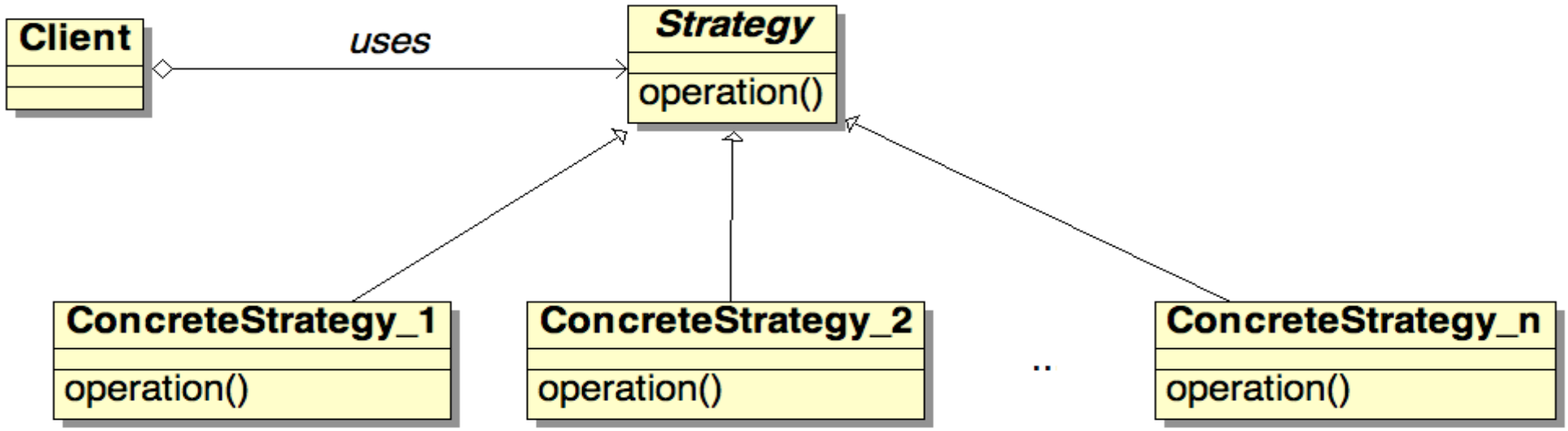
State
– defines a interface for encapsulating the behavior associated with a particular state of the Context.

ConcreteState subclasses
– each subclass implements a behavior associated with a state of the Context

Strategy (1)

- **Problème**
  - on veut (i) définir une famille d'algorithmes, (ii) encapsuler chacun et les rendre interchangeables tout en assurant que chaque algorithme peut évoluer indépendamment des clients qui l'utilisent
- **Conséquences**
  - \+ Expression hiérarchique de familles d'algorithmes, élimination de tests pour sélectionner le bon algorithme, laisse un choix d'implémentation et une sélection dynamique de l'algorithme
  - – Les clients doivent faire attention à la stratégie, surcoût lié à la communication entre Strategy et Context, augmentation du nombre d'objets

Strategy (2)



Strategy
– declares an interface common to all supported algorithms

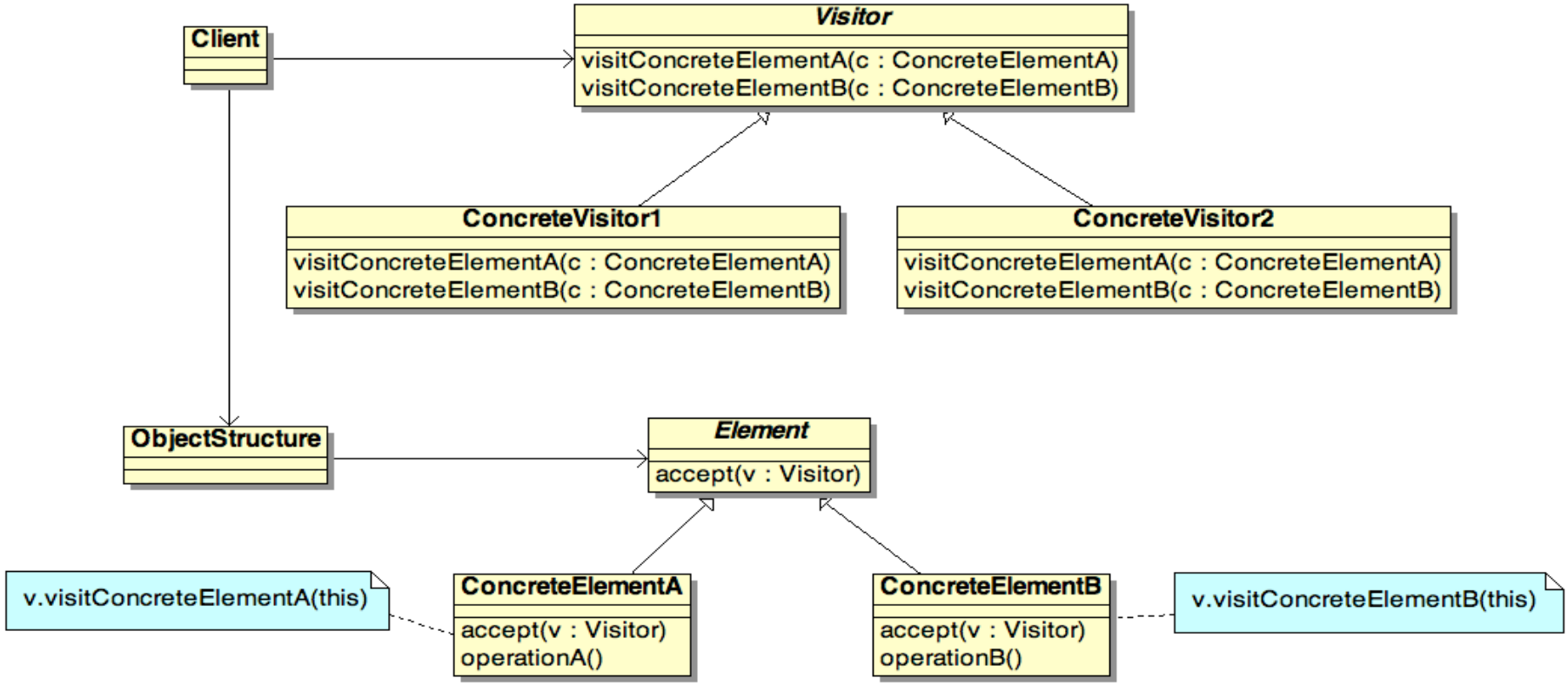
ConcreteStrategy – implements the algorithm using the Strategy interface

Client
– is configured with a ConcreteStrategy object
– maintains a reference to a Strategy object
– may define an interface that lets Strategy access its data

Visitor (1)

- **Problème**
  - Des opérations doivent être réalisées dans une structure d'objets comportant des objets avec des interfaces différentes
  - Plusieurs opérations distinctes doivent être réalisées sur des objets d'une structure
  - La classe définissant la structure change rarement mais de nouvelles opérations doivent pouvoir être définies souvent sur cette structure
- **Conséquences**
  - \+ l'ajout de nouvelles opérations est aisé
  - \+ union de différentes opérations et séparations d'autres
  - – MAIS l'ajout de nouvelles classes concrètes est freinée

Visitor (2)

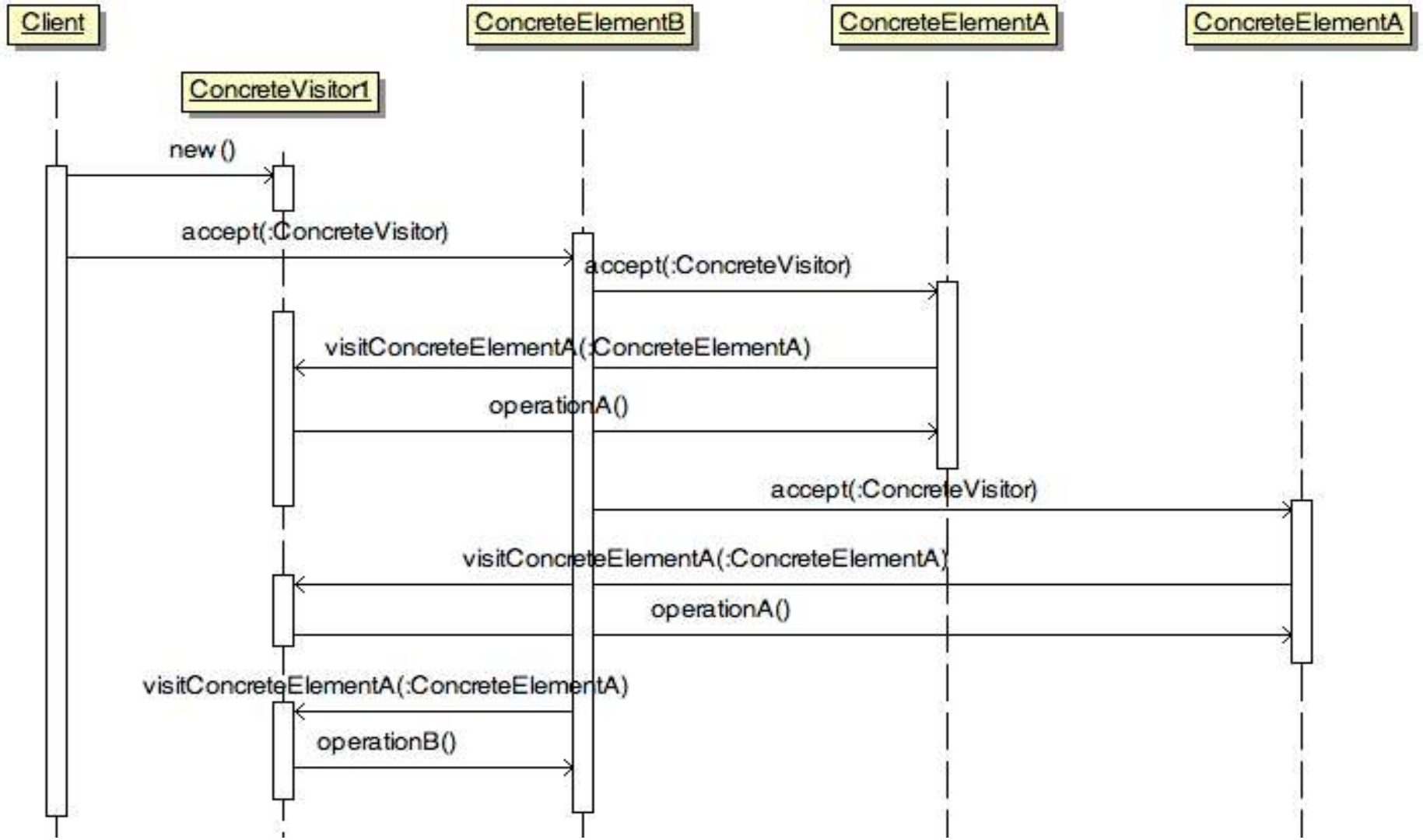


Visitor – declares a Visit operation for each class of ConcreteElement in the object structure
ConcreteVisitor – implements each operation declared by Visitor
Element – defines an Accept operation that takes a visitor as an argument

ConcreteElement – implements an Accept operation that takes a visitor as an argument

ObjectStructure
– can enumerate its elements
– may provide a high-level interface to allow the visitor to visit its elements.

Visitor (3)


Client
ConcreteElementB
ConcreteElementA
ConcreteElementA
ConcreteVisitor1
new()
accept(:ConcreteVisitor)
accept(:ConcreteVisitor)
visitConcreteElementA(:ConcreteElementA)
operationA()
accept(:ConcreteVisitor)
visitConcreteElementA(:ConcreteElementA)
operationA()
visitConcreteElementA(:ConcreteElementA)
operationB()

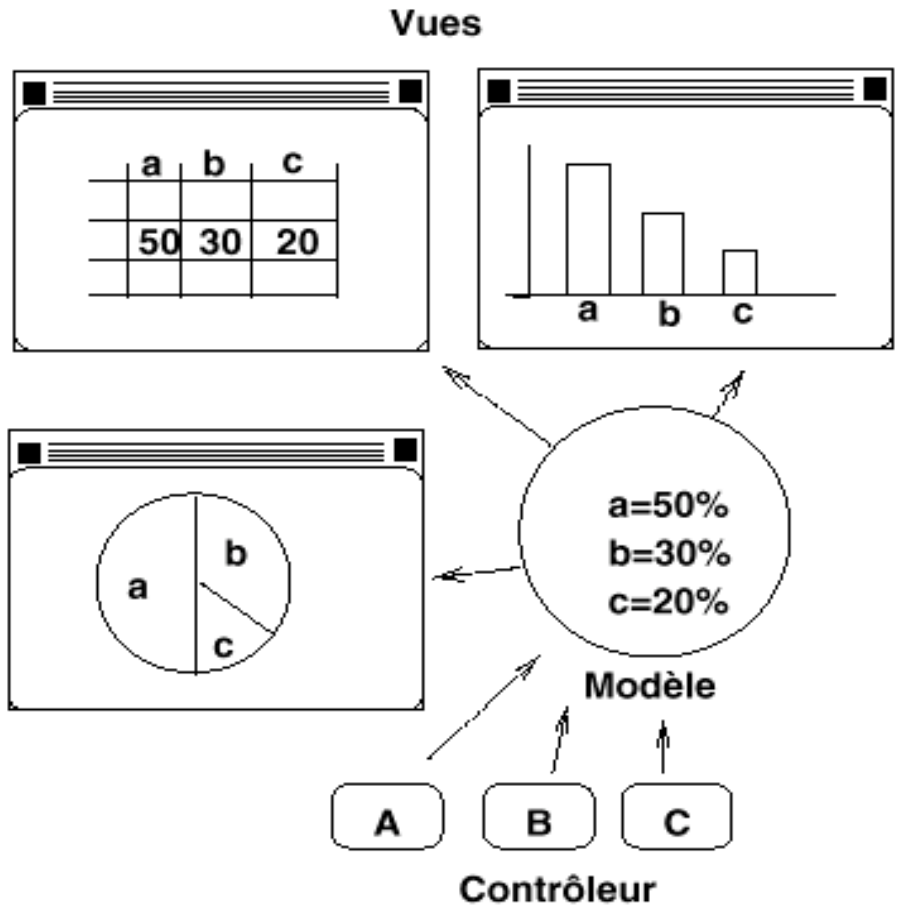
Usage et synthèse

Model View Controler

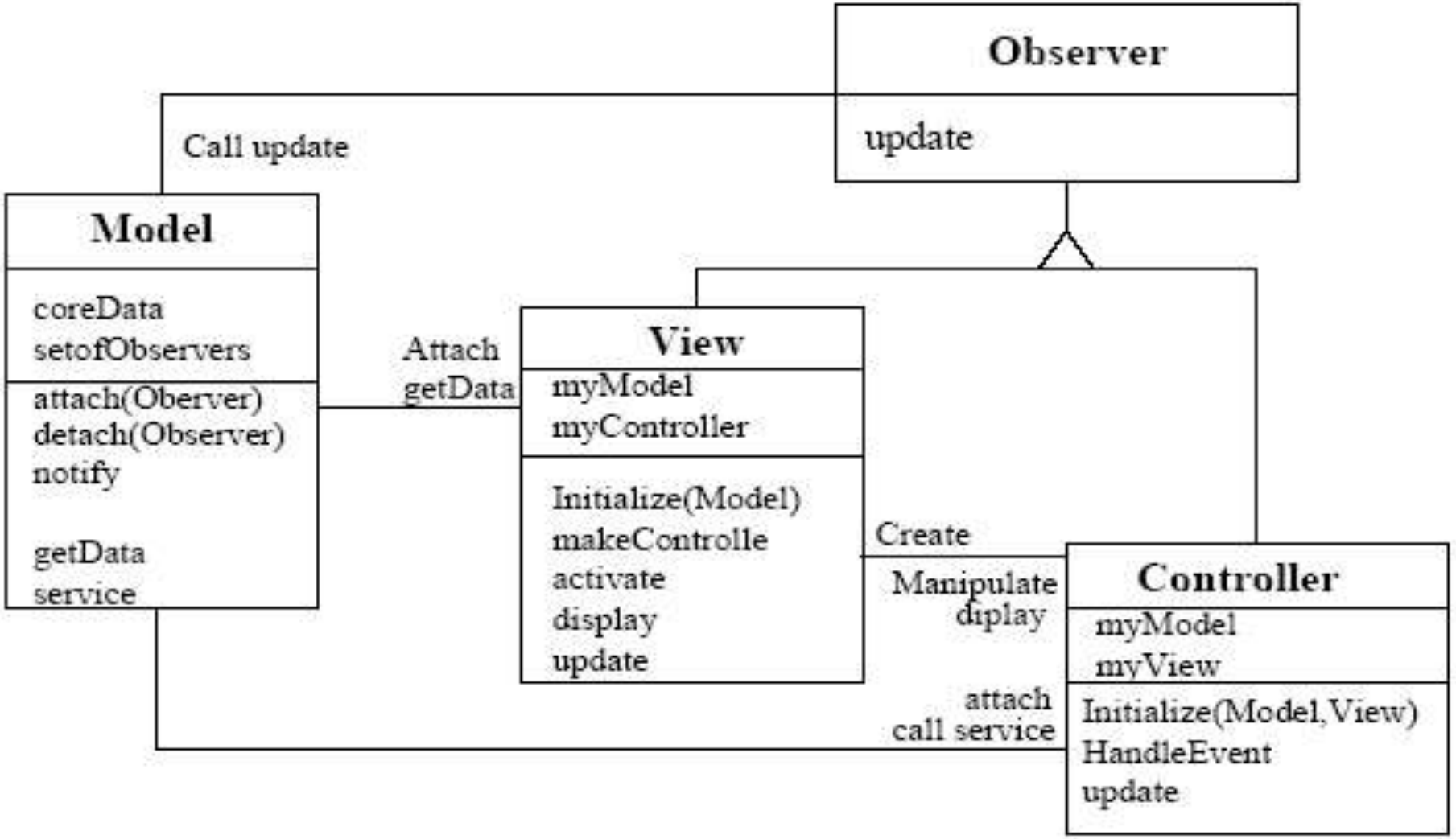
- Première version en 1980, … VisualWorks, …, Java AWT, …
- Le MVC organisé en trois types d'objets :
  - Modèle (application, pas d 'interface utilisateur),
  - Vue (fenêtres sur l'application, maintenant l 'image du modèle),
  - Contrôleur (réactions de l'interface aux actions de l'utilisateur, changement des états du modèle).
- flexibilité, réutilisation.

Model View Controler

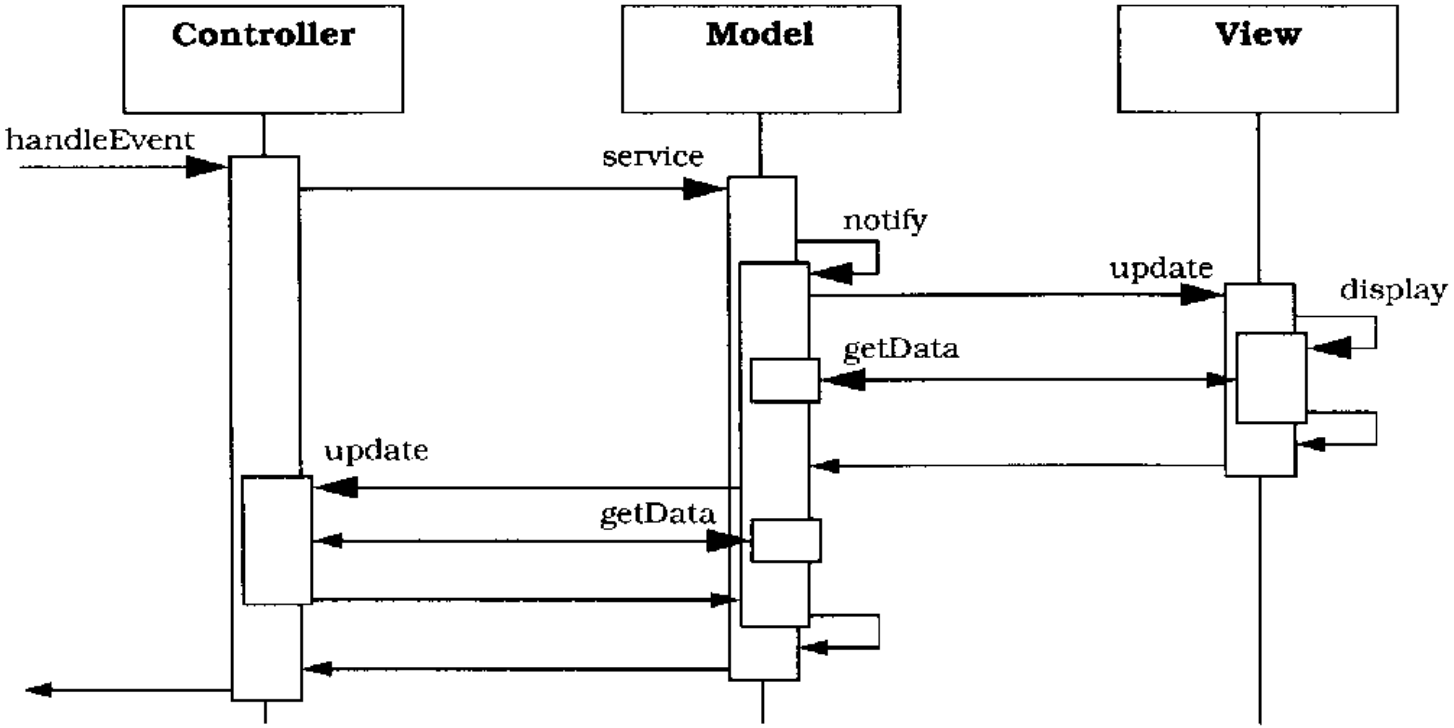
- View-View
  - Composite
  - Decorator
- View-Model
  - Observer
- View-Controler
  - Strategy
  - Factory Method
- Controler-Model
  - Command



Principe

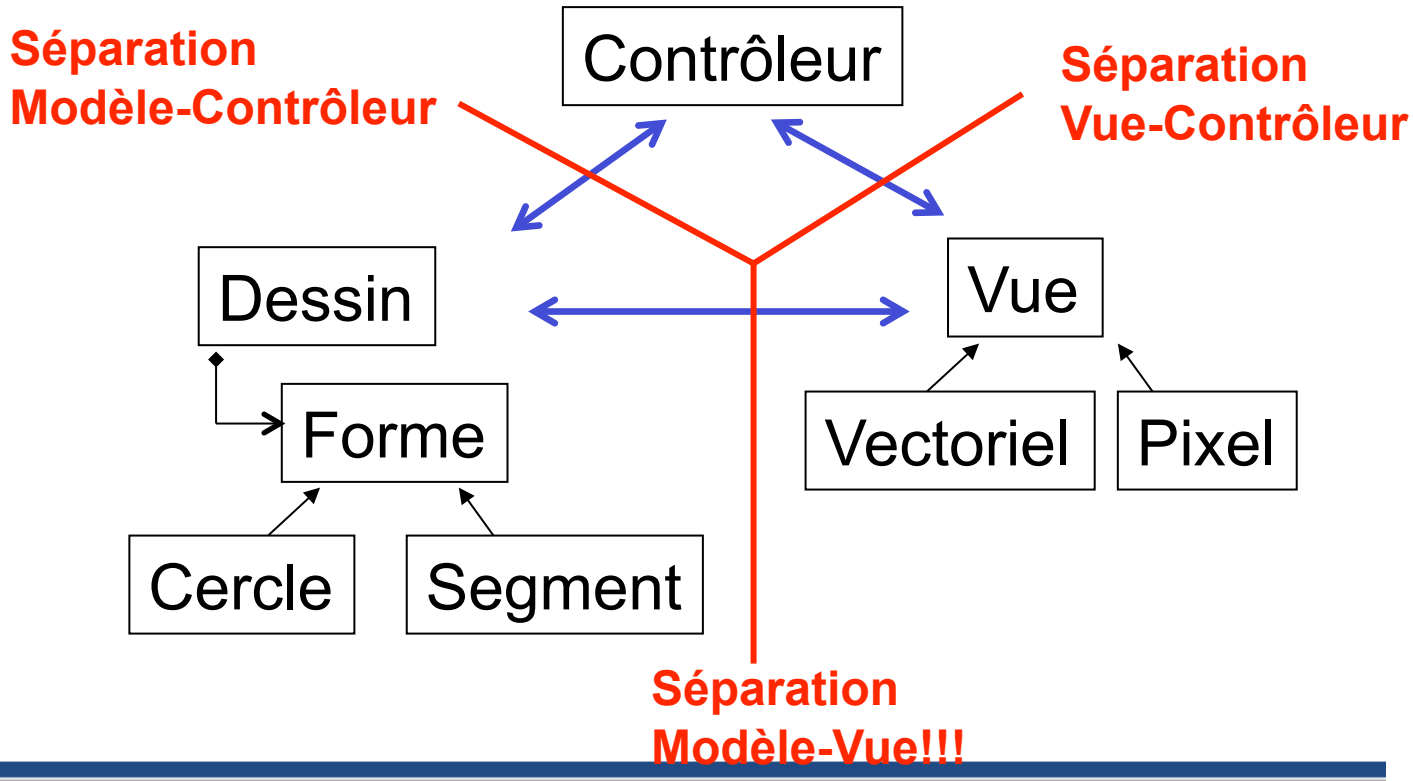

Observer
update
Call update
Model
coreData
setofObservers
attach(Oberver)
detach(Observer)
notify
getData
service
Attach
getData
View
myModel
myController
Initialize(Model)
makeControlle
activate
display
update
Create
Manipulate
diplay
attach
call service
Controller
myModel
myView
Initialize(Model,View)
HandleEvent
update

Principe

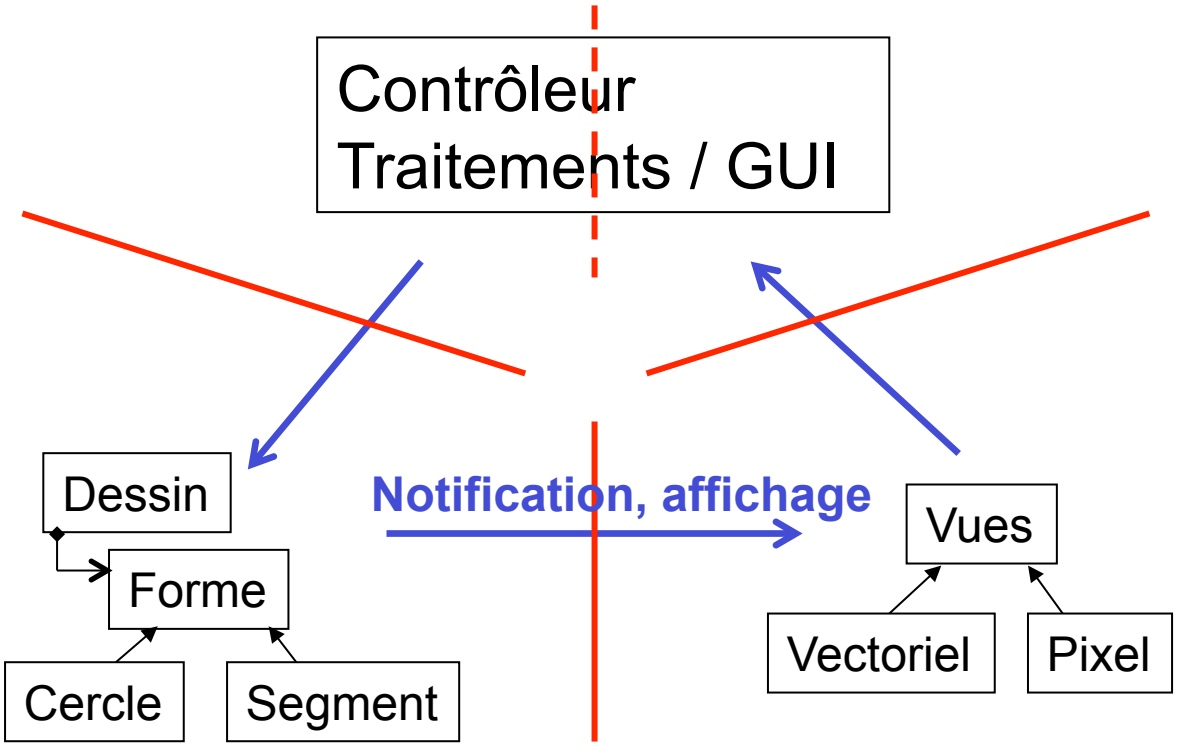

Controller
Model
View
handleEvent
service
notify
update
display
getData
update
getData

Utilisation

- Fichiers / Représentations Internes / Vues / Interactions utilisateurs

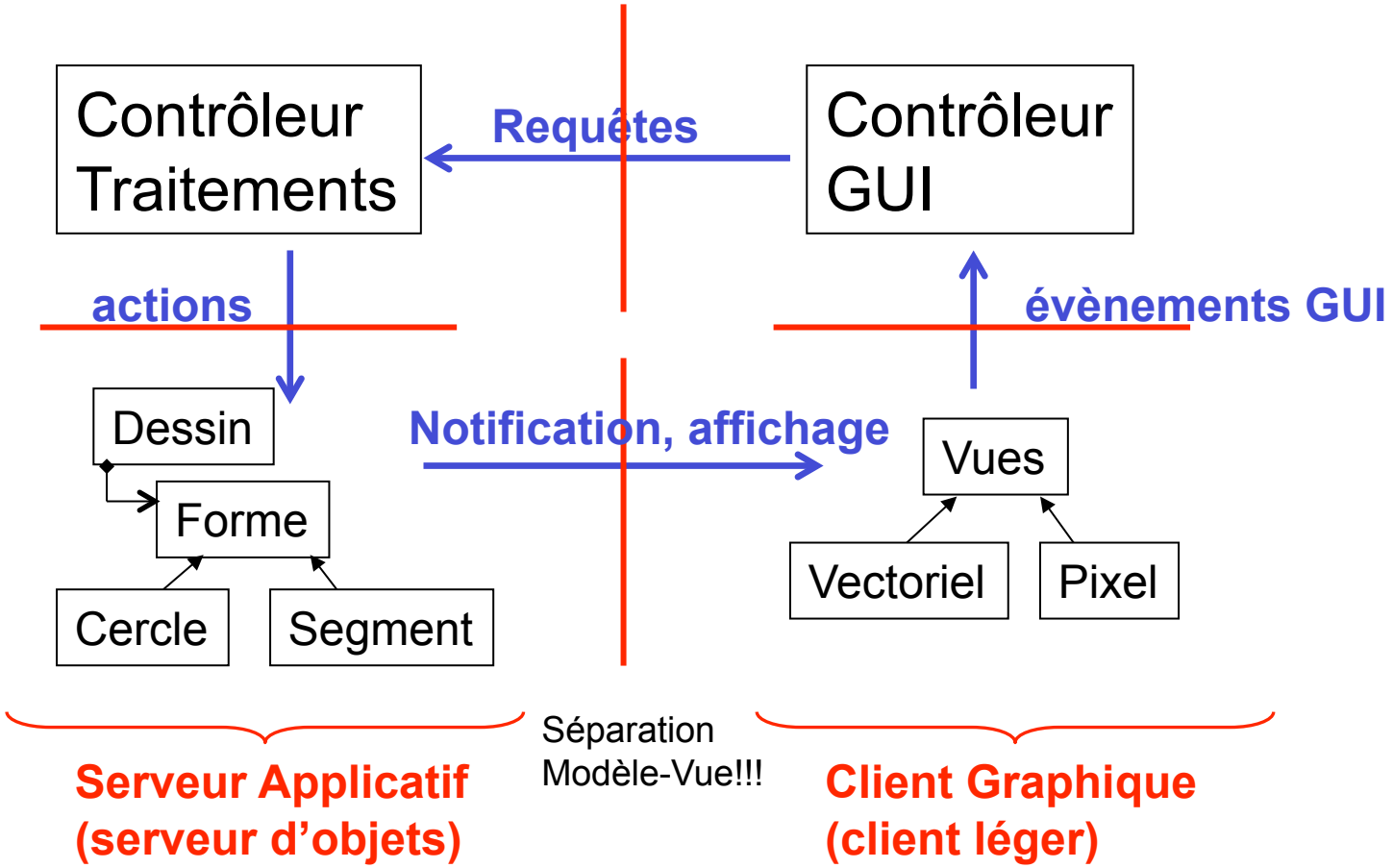


Utilisation : Contrôleur Traitements / GUI

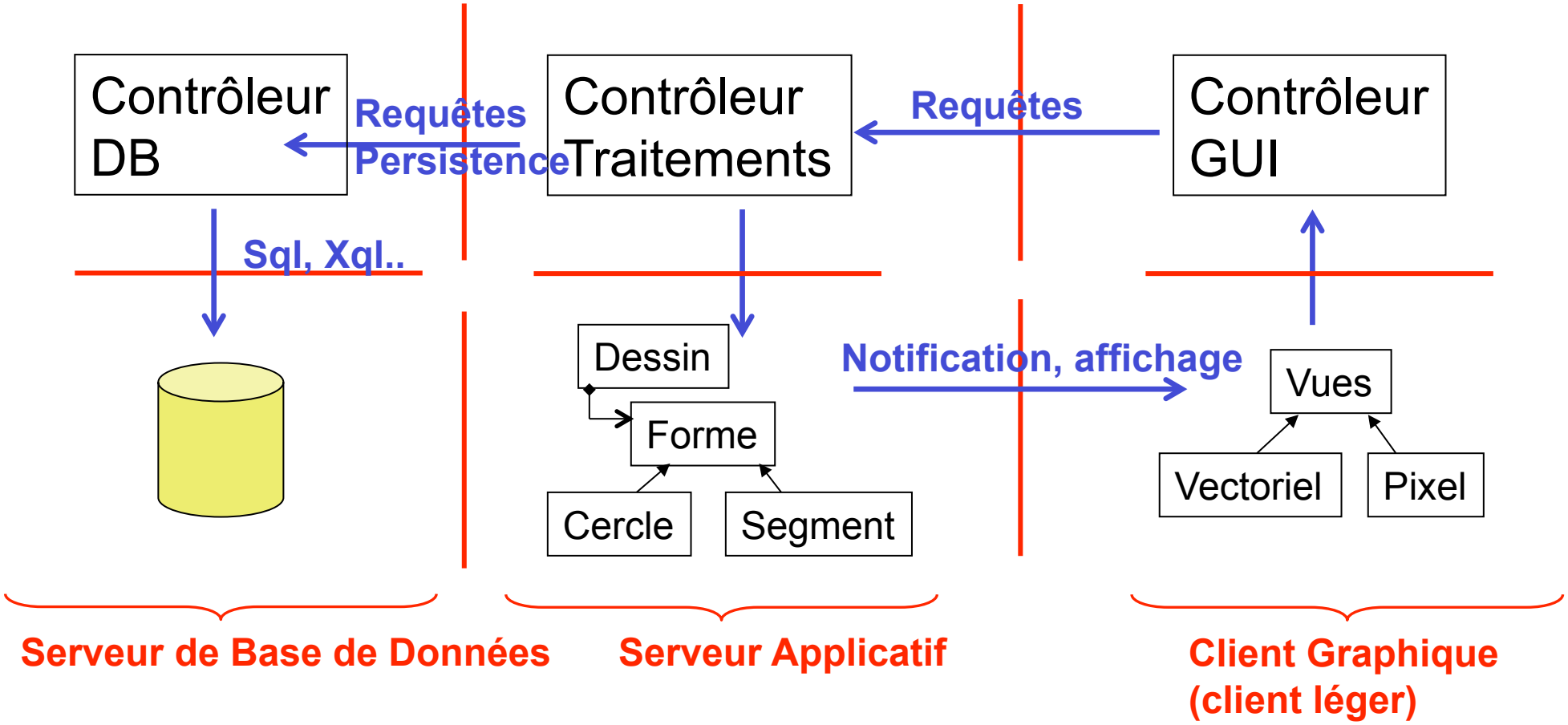


Séparation
Modèle-Vue!!!

Utilisation : Architecture 2 tiers



Utilisation : Architecture 3 tiers



Trouver les bons objets

- Les patterns proposent des abstractions qui n'apparaissent pas "naturellement" en observant le monde réel
  - Composite : permet de traiter uniformément une structure d'objets hétérogènes
  - Strategy : permet d'implanter une famille d'algorithmes interchangeables
  - State
- Ils améliorent la flexibilité et la réutilisabilité

Bien choisir la granularité

- La taille des objets peut varier considérablement
- Comment choisir ce qui doit être décomposé ou au contraire regroupé ?
  - Facade
  - Flyweight
  - Abstract Factory
  - Builder

Penser « interface »

- Qu’est-ce qui fait partie d’un objet ou non ?
  - Memento : mémorise les états, retour arrière
  - Decorator : augmente l'interface
  - Proxy : interface déléguée
  - Visitor : regroupe des interfaces
  - Facade : cache une structure complexe d'objet

Spécifier l'implémentation

- Différence type-classe...
  - Chain of Responsibility ; même interface, mais implantations différentes
  - Composite : les Components ont une même interface dont l'implantation est en partie partagée dans le Composite
  - Command, Observer, State, Strategy ne sont souvent que des interfaces abstraites
  - Prototype, Singleton, Factory, Builder sont des abstractions pour créer des objets qui permettent de penser en termes d'interfaces et de leur associer différentes implantations

Mieux réutiliser

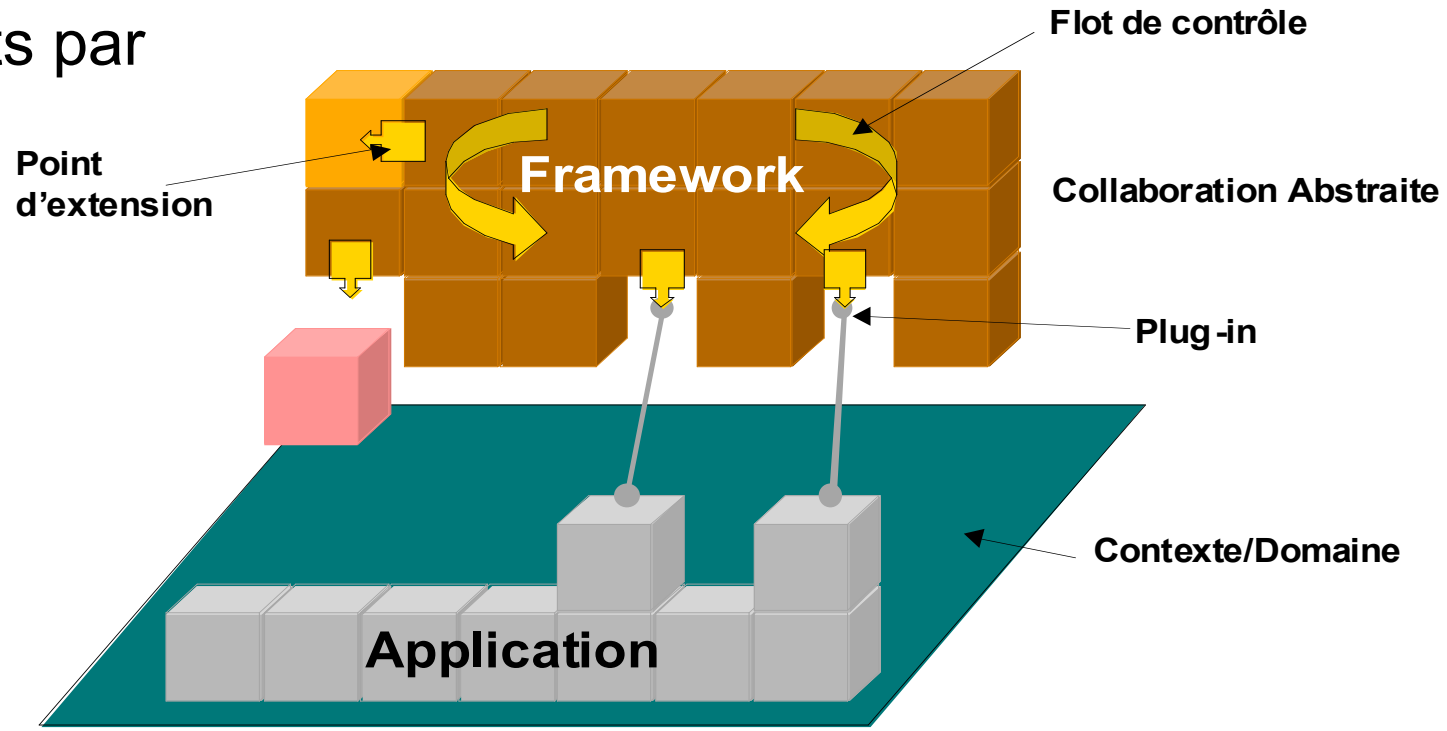
- Héritage vs Composition
  - *white-box reuse* : rompt l'encapsulation - stricte ou non
  - *black-box reuse* : flexible, dynamique

« Préférez la composition à l'héritage »

- Délégation (redirection)
  - Une forme de composition qui remplace l'héritage
  - Bridge découple l'interface de l'implantation
  - Mediator, Visitor, Proxy

Framework : cadre d'application

- Ensemble de classes :
  - concrètes ou abstraites conçues pour être utilisées ensemble, en interaction, fournissant des comportements par défaut

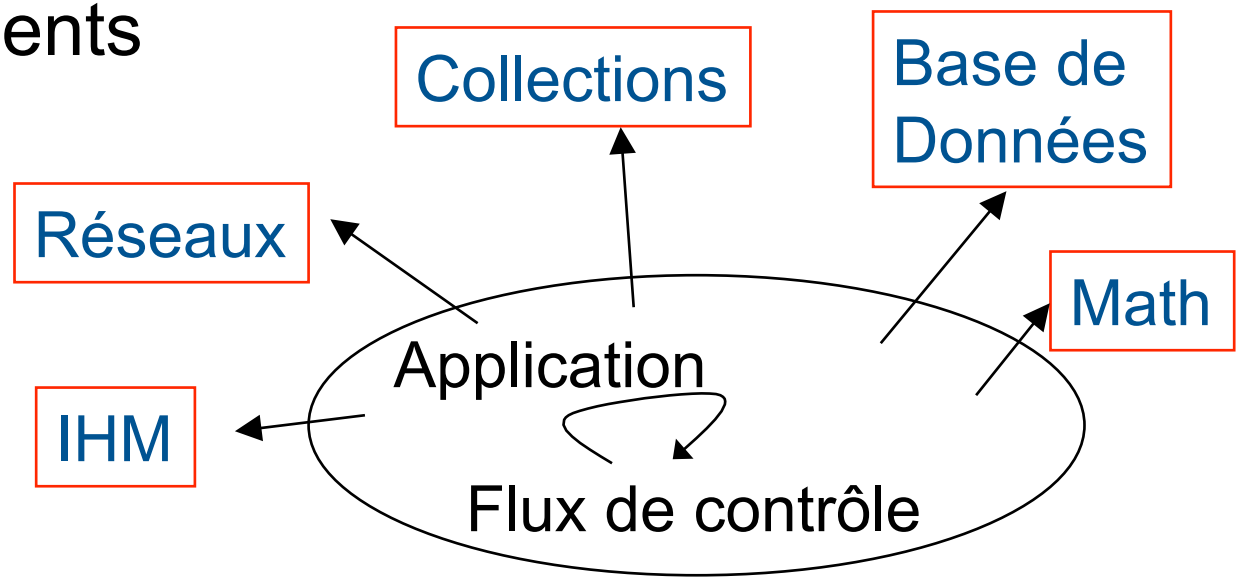


Framework : cadre d'application

- Framework d'infrastructure système :
  - pour développer des systèmes d'exploitation, des interfaces graphiques, des outils de communication. (exemple : Framework .Net, NetBeans, Struts)
  - simplification du développement d'infrastructures logicielles portables de niveau système,
  - Utilisé en interne, rarement distribué et vendu directement aux clients
- Framework d'intégration intergicielle :
  - pour fédérer des applications hétérogènes.
  - Pour mettre à dispositions différentes technologies sous la forme d'une interface unique. (exemple : ACE)
- Frameworks d'entreprise :
  - pour développer des applications spécifiques au secteur d'activité de l'entreprise.
- Frameworks orientés Système de gestion de contenu

Framework vs Bibliothèque de classes

- Ensemble de classes
  - le plus souvent abstraites
  - indépendantes (pas d ’interaction a priori entre elles)
  - sans comportements par défaut



Framework : exemples

- Aoof-Wm — un framework multiplateforme pour la creation de Window Manager, open-source écrit en C++ (Homepage GoogleProject)
- Apache Cocoon — un environnement servlet de développement JAVA/XML pour le web, fondation Apache Software
- Apache Struts — de la fondation Apache Software
- Catalyst (logiciel) — un framework web open-source écrit en langage de programmation Perl
- Cocoa — de la société Apple
- Django — un framework web open-source écrit en langage de programmation Python
- Dojo -- développement rapide d'applications en Javascript exécutées côté butineur et communiquant avec le serveur avec une granularité inférieure à la page grâce à Ajax.
- EVAcms — de la société EVA Soluções
- NetBeans — de la société Sun Microsystems
- Lampshade (Framework) — de la société Think Computer
- Microsoft .NET — de la société Microsoft
- Maypole framework — un framework web open-source écrit en Perl
- MIReG - Framework de métadonnées de l'Union européenne
- RIFE — un framework web open-source écrit en langage de programmation Java
- Ruby on Rails — un framework web open-source écrit en langage de programmation Ruby
- Seaside — un framework web open-source écrit en Smalltalk
- TurboGears — un framework web open-source écrit en langage de programmation Python
- Zope — un framework web open-source écrit en langage de programmation Python
- XNA — un framework de jeu vidéo écrit en C#. Il permet le développement d'application pour plateformes Xbox 360 & PC
- GLOBO - un framework pour le développement d'applications de back office dans un environnement ORACLE

Source Wikipedia

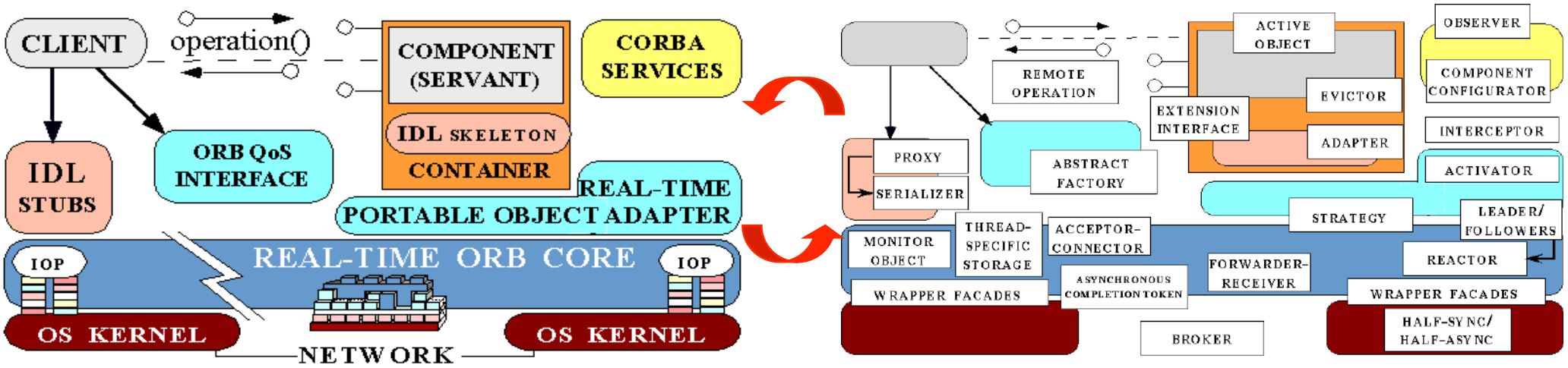
Extension de frameworks

- ***Framework de type "Boîte blanche"*** :
  - conçu par la généralisation de classes de différentes applications,
  - étendu par :
    (1) dérivation de classes de base
    (2) surcharge de méthodes des classes dérivées
- ***Framework de type "Boîte noire"*** :
  - conçu par composition,
  - étendu par composition des composants entre eux.
    (1) implémentation des interfaces de composants intégré dans le framework par composition
    (2) intégration de ces composants dans le framework

Design Patterns / Frameworks

- Synergie entre les concepts de Design Patterns et de Frameworks
- les Design Patterns :
  - décrivent à un niveau plus abstrait des composants de frameworks, implémentés dans un langage particulier,
  - aident à la conception des frameworks,
- la conception de Frameworks
  - aide à la découverte de nouveaux design patterns.

Design Patterns / Frameworks

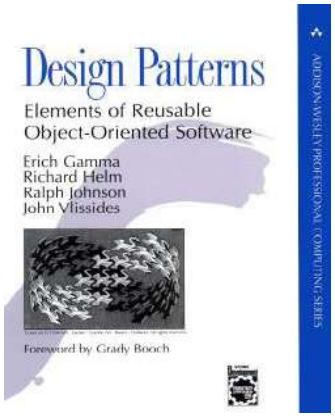


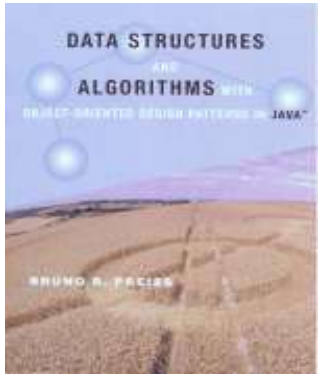
tiré de cours D.C. Schmidt

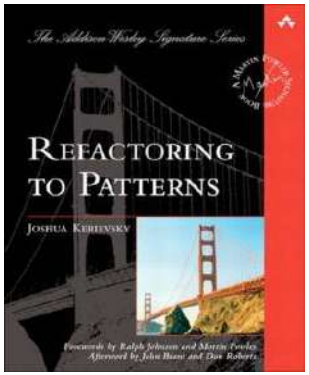
En bref, les Design Patterns…

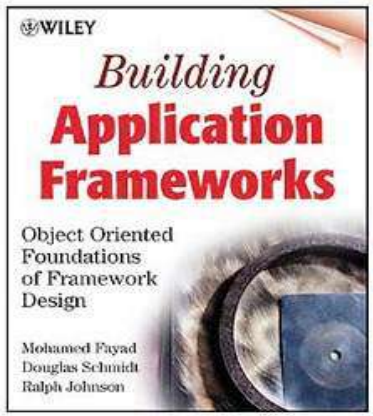
- C’est...
  - une description d’une solution classique à un problème récurent
  - une description d’une partie de la solution... avec des relations avec le système et les autres parties...
  - une technique d’architecture logicielle
- Ce n’est pas...
  - une brique
    - Un pattern dépend de son environnement
  - une règle
    - Un pattern ne peut pas s’appliquer mécaniquement
  - une méthode
    - Ne guide pas une prise de décision : un pattern est la décision prise

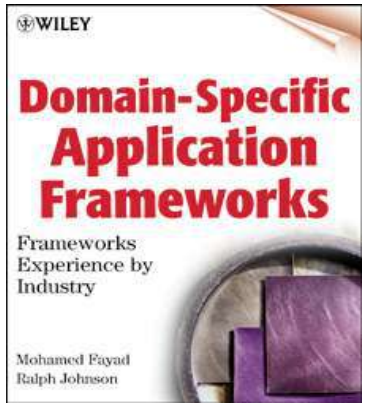
Bibliographie

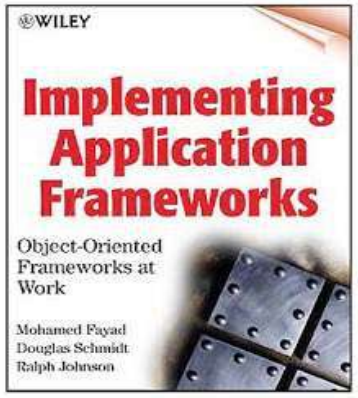


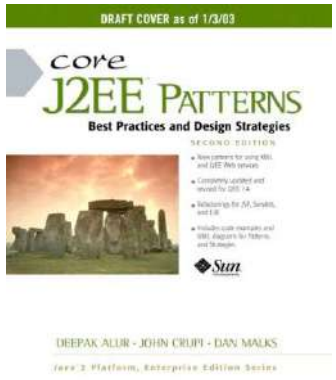












Bibliographie

- ” Pattern Languages of Program Design ”, Coplien J.O., Schmidt D.C., Addison-Wesley, 1995.
- ” Pattern languages of program design 2 ”, Vlissides, et al, ISBN 0-201-89527-7, Addison-Wesley
- ” Pattern-oriented software architecture, a system of patterns ”, Buschmann, et al, Wiley
- ” Advanced C++ Programming Styles and Idioms ”, Coplien J.O., Addison-Wesley, 1992.
- S.R. Alpert, K.Brown, B.Woolf (1998) The Design Patterns Smalltalk Companion, Addison-Wesley (Software patterns series).
- J.W.Cooper (1998), The Design Patterns Java Companion, http:// www.patterndepot.com/put/8/JavaPatterns.htm.
- S.A. Stelting, O.Maasen (2002) Applied Java Patterns, Sun Microsystems Press.
- Communications of ACM, October 1997, vol. 40 (10).
- Thinking in Patterns with Java http://mindview.net/Books/TIPatterns/

Quelques sites Web

- http://hillside.net/
- Portland Pattern Repository
  - http://www.c2.com/ppr
- A Learning Guide To Design Patterns
  - http://www.industriallogic.com/papers/learning.html
- Vince Huston
  - http://home.earthlink.net/~huston2/
- Ward Cunnigham's WikiWiki Web
  - http://www.c2.com/cgi/wiki?WelcomeVisitors
- Core J2EE Patterns
  - http://www.corej2eepatterns.com/index.htm