UNIVERSITÉ DE RENNES 1

ESIR ECOLE SUPERIEURE D'INGENIEURS DE RENNES

# Méthodes de Développement Industriel (MDI)

**Mathieu Acher**

http://www.mathieuacher.com

*Associate Professor*

*University of Rennes 1*

UMR IRISA

Inria informatics mathematics

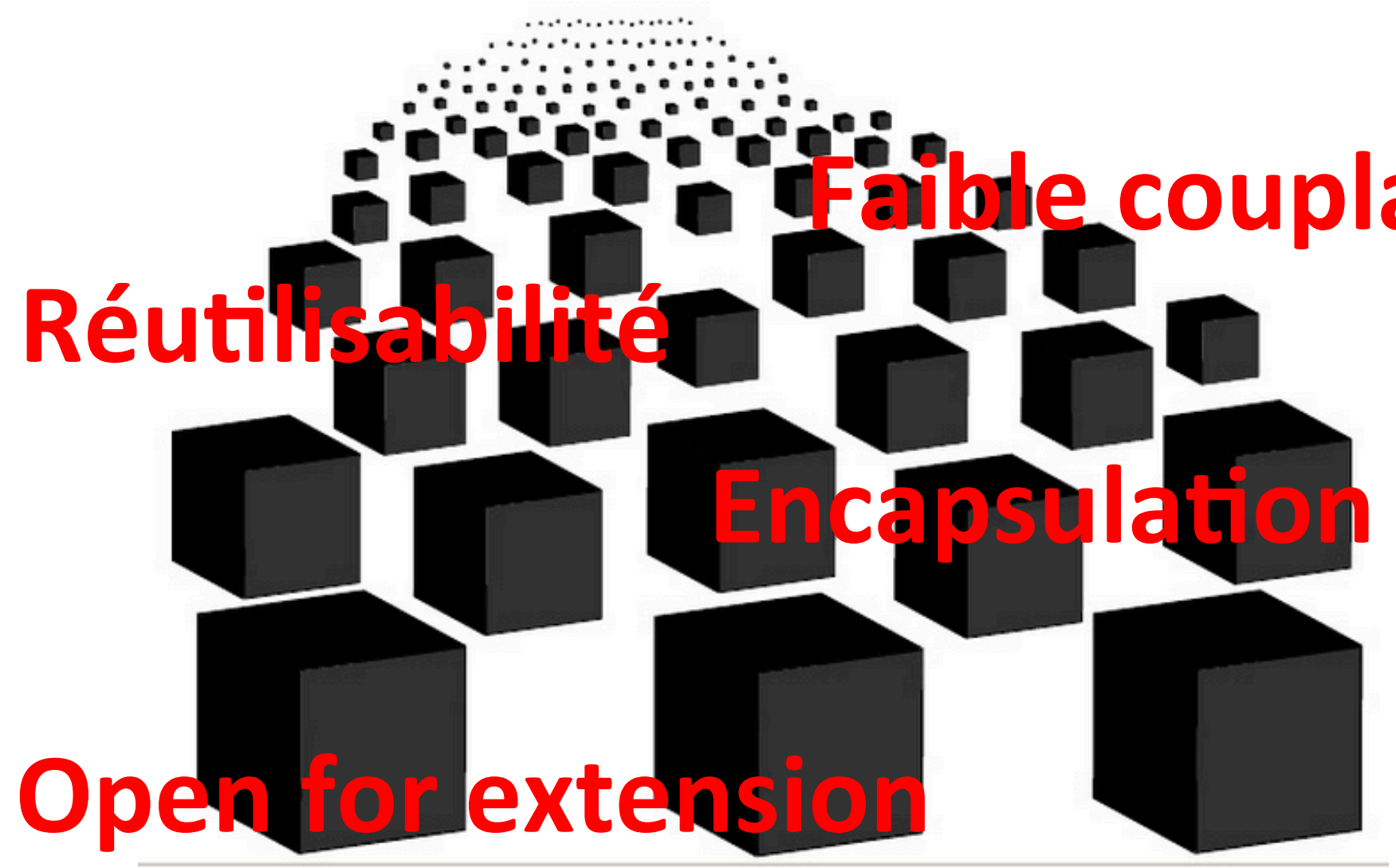
# Objectifs de MDI

- Méthodes de développement industriel (MDI)
  - En fait: génie logiciel / software engineering
  - Comment développer des systèmes logiciels de plus en plus complexe?
- #1 Prendre conscience de la complexité des systèmes logiciels actuels et à venir
  - Les enjeux et l'impact sur le métier
- #2 Modélisation
  - UML, SysML
- **#3 Design patterns, refactoring, test**
  - **OO avancé**
- #4 Méthodes

# Design Patterns

## Motivation, Principes

# Idéalement:
# « modular black boxes »



Faible couplage

Réutilisabilité

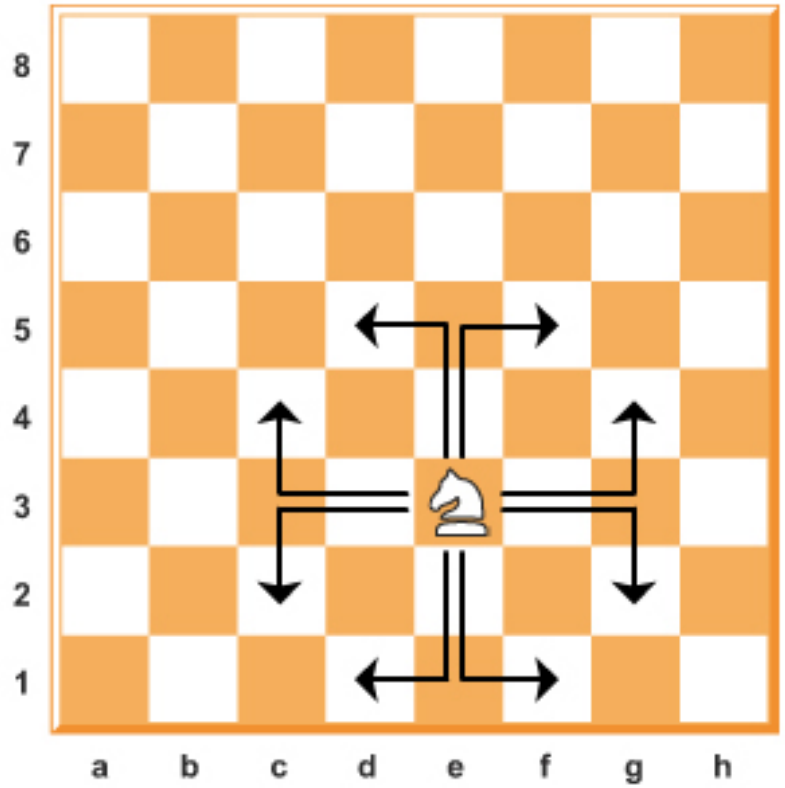
Encapsulation

Open for extension

# Tout le contraire d'un code monolithique (une seule classe)

**Version graphique?**
**Sauvegarder/Charger une partie?**
**Comment s'assurer que la règle du pat a bien été prise en compte?**
**Nouveau moteur de AI?**

- ChessDay1
  - main(String[]) : void
  - board : PieceType[][]
  - ChessDay1()
  - canMove(int, int, int, int) : boolean
  - displayBoard() : void
  - get50MoveRulePlyCount() : int
  - getCapturedPieces() : List<PieceType>
  - getCapturedPieces(boolean) : List<PieceType>
  - getCurrentMoveNumber() : int
  - getHistory() : String
  - getMaterialCount(boolean) : int
  - getPossibleMoves() : List<int[][]>
  - getPossibleSquares(int, int) : List<int[][]>
  - getThreats(int, int) : List<int[][]>
  - getUnCapturedPieces(boolean) : List<PieceType>
  - initBoard() : void
  - isBlackCastleableKingside() : boolean
  - isBlackCastleableQueenside() : boolean
  - isBlockable() : boolean
  - isCheck() : boolean
  - isCheckmate() : boolean
  - isDoubleCheck() : boolean
  - isEnPassantFile(int) : boolean
  - isStalemate() : boolean
  - isWhiteCastleableKingside() : boolean
  - isWhiteCastleableQueenside() : boolean
  - isWhiteToPlay() : boolean
  - recordMove(int, int, int, int) : boolean
  - redo() : boolean
  - reset() : void
  - setBlackCastleableKingside(boolean) : void
  - setBlackCastleableQueenside(boolean) : void
  - setWhiteCastleableKingside(boolean) : void
  - setWhiteCastleableQueenside(boolean) : void
  - undo() : boolean

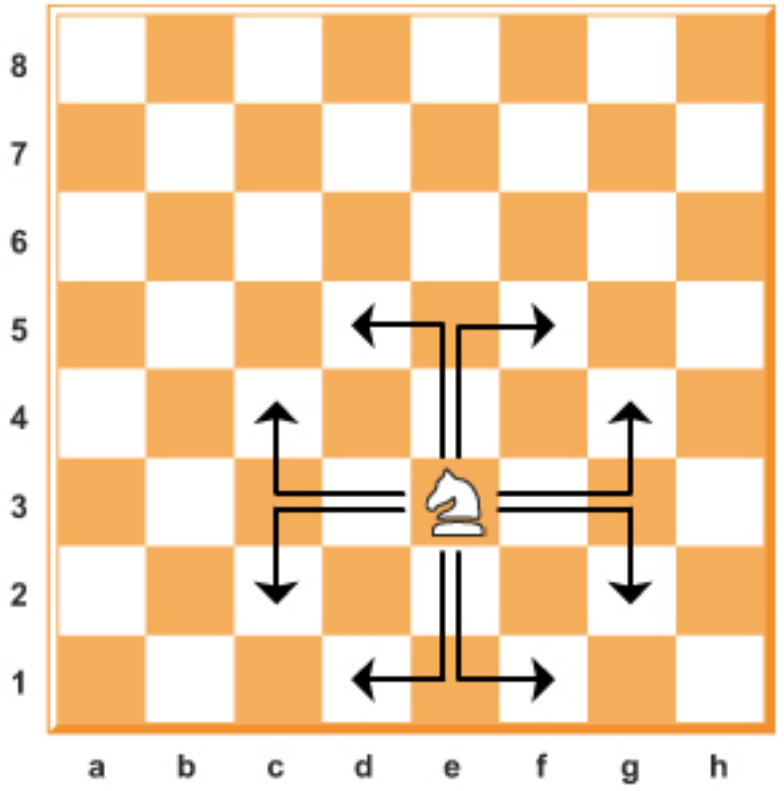



# fig.move(“c2”);

**Comment créer et initaliser l’échiquier ?**
**Comment implémenter la logique des coups/règles?**
**….**

- ChessDay1
  - main(String[]) : void
  - board : PieceType[][]
  - ChessDay1()
  - canMove(int, int, int, int) : boolean
  - displayBoard() : void
  - get50MoveRulePlyCount() : int
  - getCapturedPieces() : List<PieceType>
  - getCapturedPieces(boolean) : List<PieceType>
  - getCurrentMoveNumber() : int
  - getHistory() : String
  - getMaterialCount(boolean) : int
  - getPossibleMoves() : List<int[][]>
  - getPossibleSquares(int, int) : List<int[][]>
  - getThreats(int, int) : List<int[][]>
  - getUnCapturedPieces(boolean) : List<PieceType>
  - initBoard() : void
  - isBlackCastleableKingside() : boolean
  - isBlackCastleableQueenside() : boolean
  - isBlockable() : boolean
  - isCheck() : boolean
  - isCheckmate() : boolean
  - isDoubleCheck() : boolean
  - isEnPassantFile(int) : boolean
  - isStalemate() : boolean
  - isWhiteCastleableKingside() : boolean
  - isWhiteCastleableQueenside() : boolean
  - isWhiteToPlay() : boolean
  - recordMove(int, int, int, int) : boolean
  - redo() : boolean
  - reset() : void
  - setBlackCastleableKingside(boolean) : void
  - setBlackCastleableQueenside(boolean) : void
  - setWhiteCastleableKingside(boolean) : void
  - setWhiteCastleableQueenside(boolean) : void
  - undo() : boolean

**Comment créer et initaliser l'échiquier ?**
**Comment implémenter la logique des coups/règles?**
**Comment sauvegarder une partie?**
**Comment supporter plusieurs formats?**
**Comment intégrer de nouveaux moteurs?**
**Comment changer l'apparence de l'échiquier (à l'exécution)?**
**…**




**Solutions existantes à des problèmes déjà résolus?**

**Patterns (motifs, patrons)**

"Each **pattern** describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" – Christopher Alexander

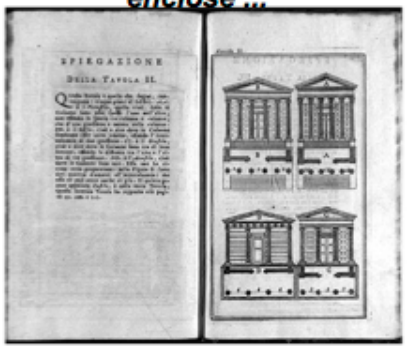
# Clues from other disciplines *(from E. Gamma)*

## architecture

***"I have drawn up definite rules to enable you to have personal knowledge of the quality both of existing buildings and of those which are yet to be constructed."***

***...***

***"A temple is called IN ANTIS, when it has antæ or pilasters in front of the walls which enclose ..."***

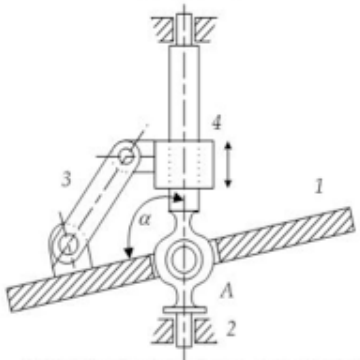


**Marcus Vitruvius Pollio**
***"De Architectura Libri Decem" 27 BC***

## mechanical engineering handbooks

***"Mechanisms of Modern Engineering Design": Ivan Artobolevsky 1947***

Slider Crank Mechanism of a Centrifugal Governor
1636 SC:G

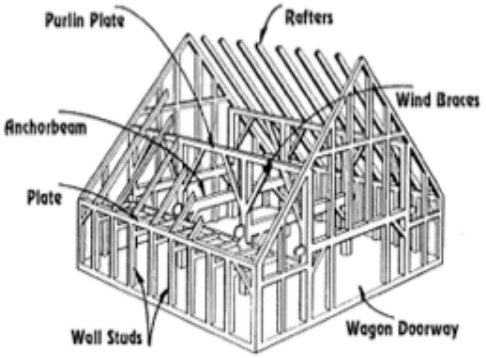



Link *1* is designed as a round plate turning about axis *A*. The angle α between the plane and the axis of rotation of shaft 2 depends upon the centrifugal force. When angle α is changed, connecting rod *3* slides sleeve 4 along the axis of shaft 2.

## furniture and timber framing handbooks

***"If you combine technique and knowledge of the material, will automatically design around the construction, and not construct around the design. ... as construction becomes second nature when you are designing."***

**Tage Frid**

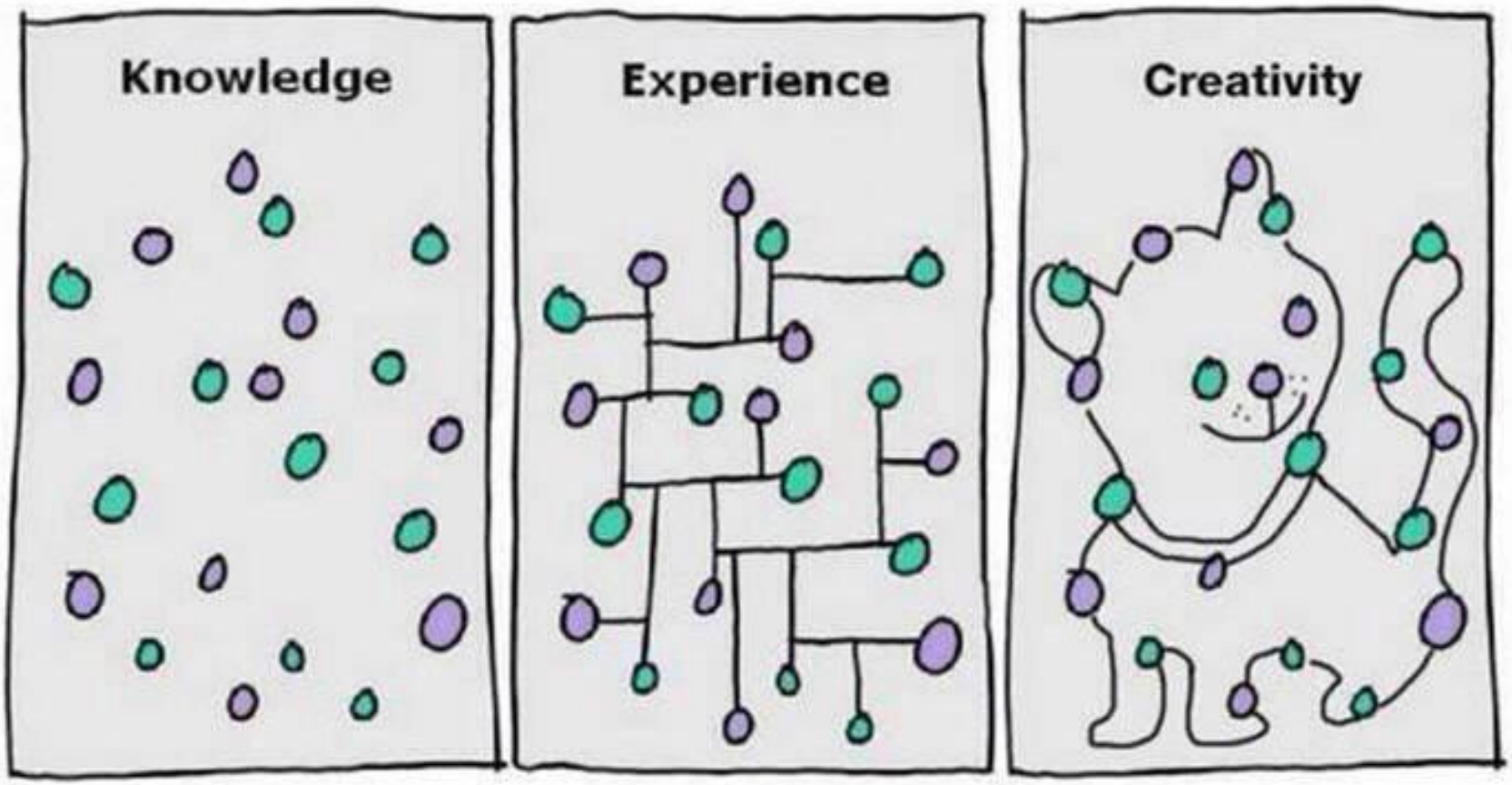



*cf.* [http://fose.ethz.ch/slides/gamma.pdf](http://fose.ethz.ch/slides/gamma.pdf)

# Patterns in Physical Architecture

- When a room has a window with a view, the window becomes a focal point: people are attracted to the window and want to look through it. The furniture in the room creates a second focal point: everyone is attracted toward whatever point the furniture aims them at (usually the center of the room or a TV). This makes people feel uncomfortable. They want to look out the window, and toward the other focus at the same time. If you rearrange the furniture, so that its focal point becomes the window, then everyone will suddenly notice that the room is much more "comfortable"

Leonard Budney, Amazon.com review of *The Timeless Way of Building*


Knowledge
Experience
Creativity

# Design Pattern : Pourquoi ?

- Validation qualitative des acquis et de la connaissance pratique
  - Vers une « ingénierie » (caractère systématique) du logiciel
  - Faciliter la réutilisation de savoir faire
  - Identifier, comprendre et appréhender des solutions récurrentes (e.g., API, framework existant)
- Indispensable pour
  - Comprendre l'existant
  - Réutiliser / étendre / tester l'existant
  - Construire de nouveaux systèmes logiciels

# Sur la réutilisation...

- Les langages informatiques modernes orientés objet permettent la réutilisation
  - par importation de classes
  - par héritage : extension / spécialisation
  - par l'inversion de contrôle (aspects)

# Design Pattern : C'est quoi ?

- Un fragment d'architecture à objets
- Une solution « classique » à un problème fréquent
- Une solution indépendante des algorithmes
- Une solution qui découple les différents problèmes et leurs différentes réponses

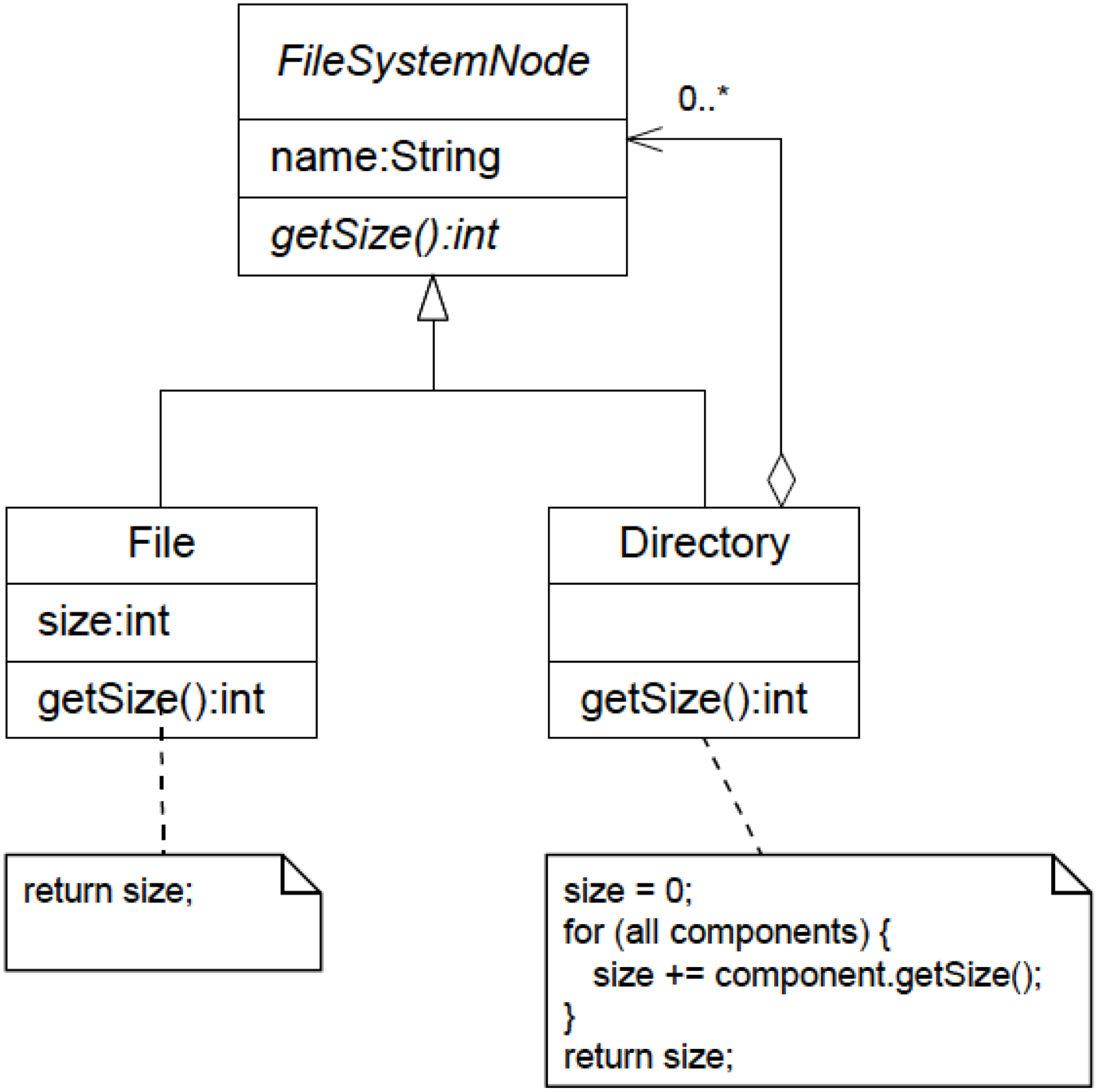
# Design Pattern : Origine ?

- Concept proposé pour les architectures de bâtiments (Christopher Alexander)
- Début d'application aux architectures logicielles en 1987
- Visibilité publique en 1994 grâce au livre
  - Design patterns: elements of reusable object-oriented software (Gamma, Helm, Johnson et Vlissides, dit le Gang of Four : GoF)

- Context: File System
  - Files, directories
- Goal: Compute the number of files in the file system

- Have you got a pattern? (a reusable solution to the problem)

- Solution in…
  - UML
  - Java


FileSystemNode
name:String
getSize():int
0..*
File
size:int
getSize():int
Directory
getSize():int
return size;
size = 0;
for (all components) {
size += component.getSize();
}
return size;

# Elements of a Pattern

- Name
- Problem
  - When the pattern is applicable
- Solution
  - Design elements and their relationships
  - Abstract (must be specialized/instantiated)
- Consequences
  - Cost versus benefits
  - Flexibility, extensibility
  - Variations in the pattern may imply difference consequences

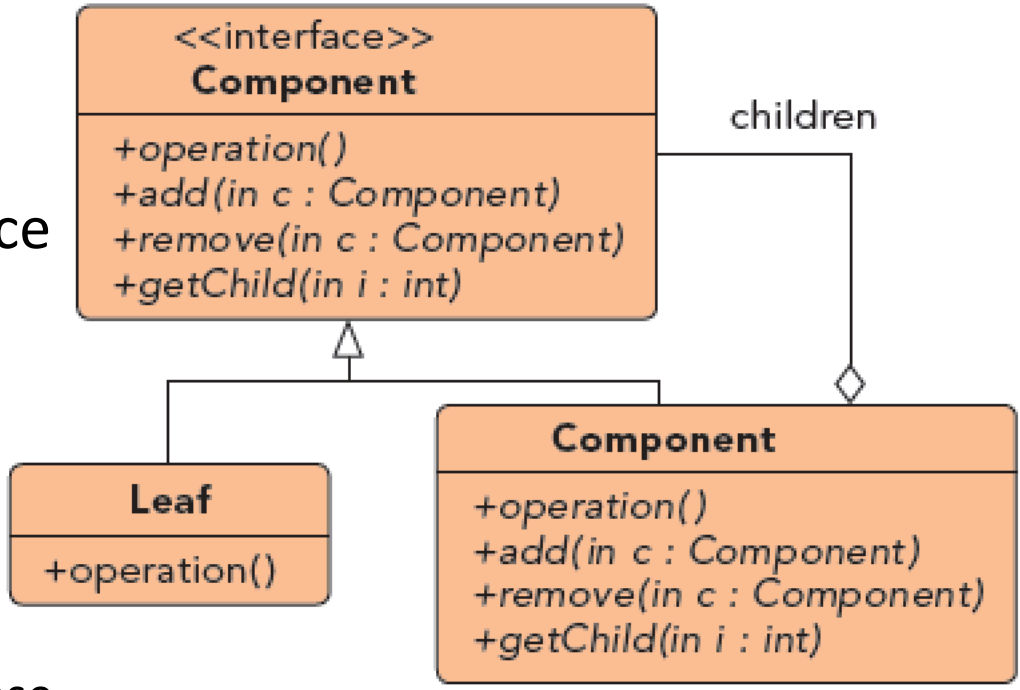
# Design Pattern : Catégories ?

- Patrons de création
  - ils ont pour but de gérer les problèmes de création de nouveaux objets
    - *Abstract Factory, Builder, Factory Method, Prototype, Singleton*
- Patrons de structure
  - ils servent à organiser les informations dans un graphe d'objets
    - *Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy*
- Patrons de comportement
  - ils servent à maîtriser les interactions entre objets
    - *Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor, Callback*

# Composite

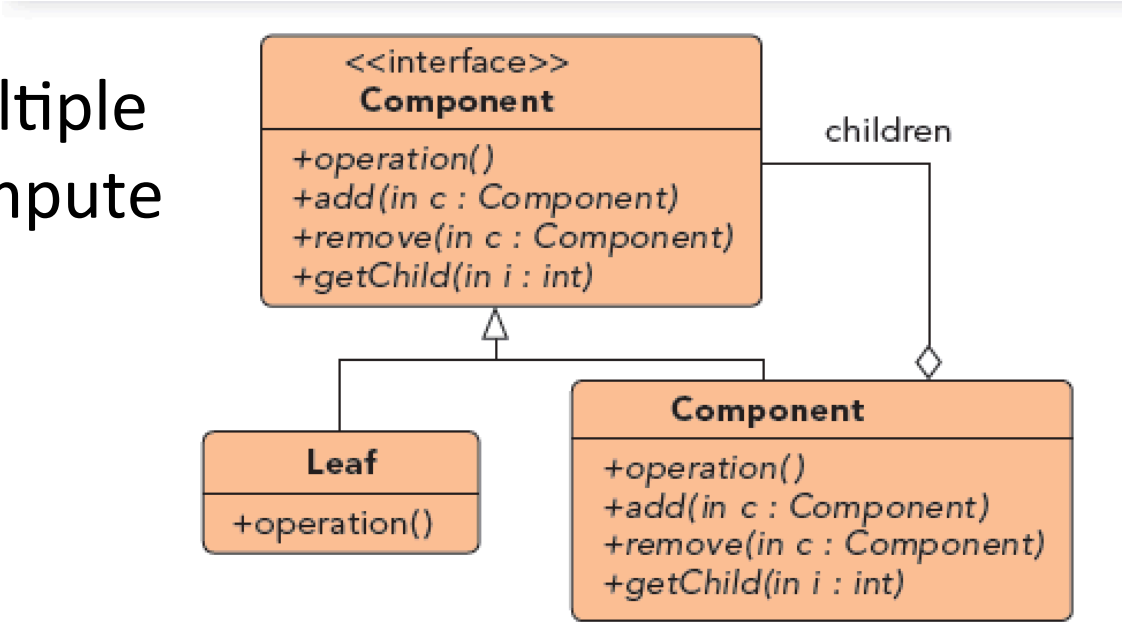
# Composite

- Applicability
  - You want to represent part-whole hierarchies of objects
  - You want to be able to ignore the difference between compositions of objects and individual objects
- Consequences
  - Makes the client simple, since it can treat objects and composites uniformly
  - Makes it easy to add new kinds of components
  - Can make the design overly general
    - Operations may not make sense on every class
    - Composites may contain only certain components

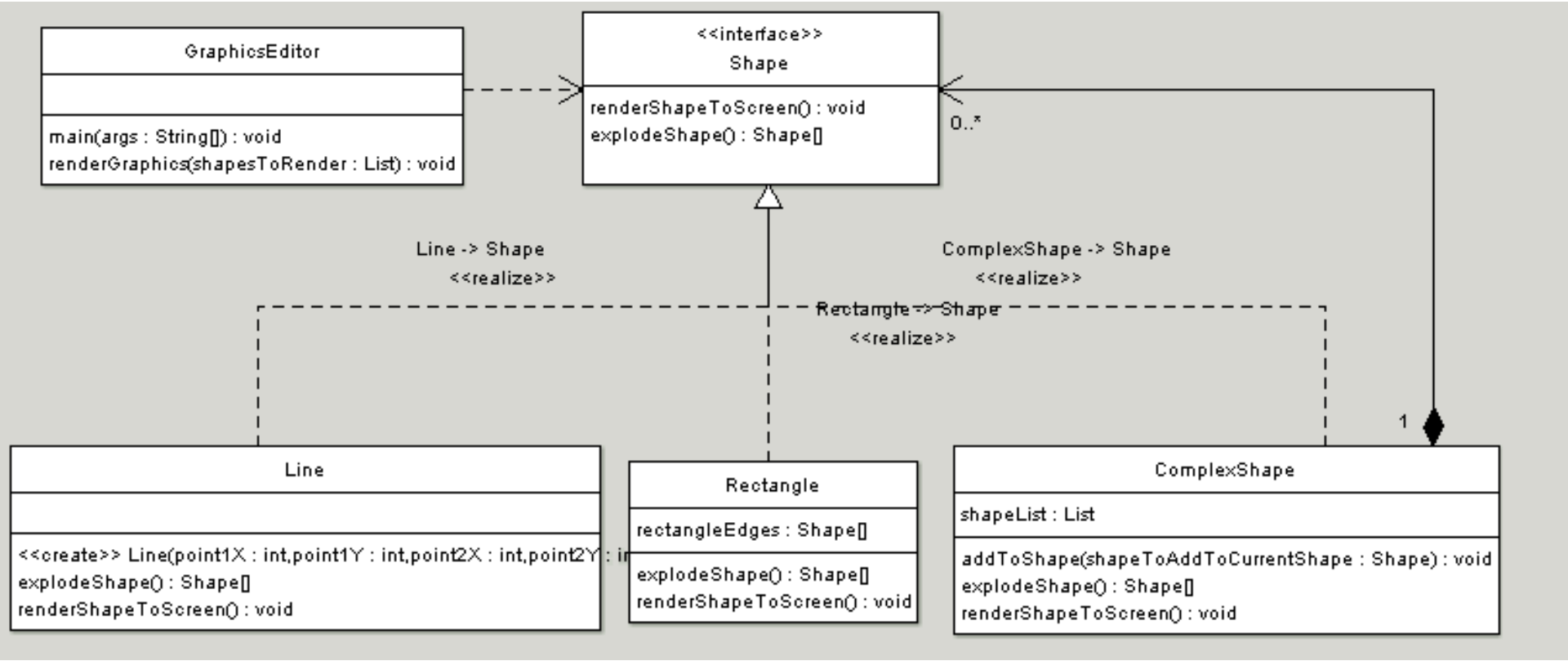


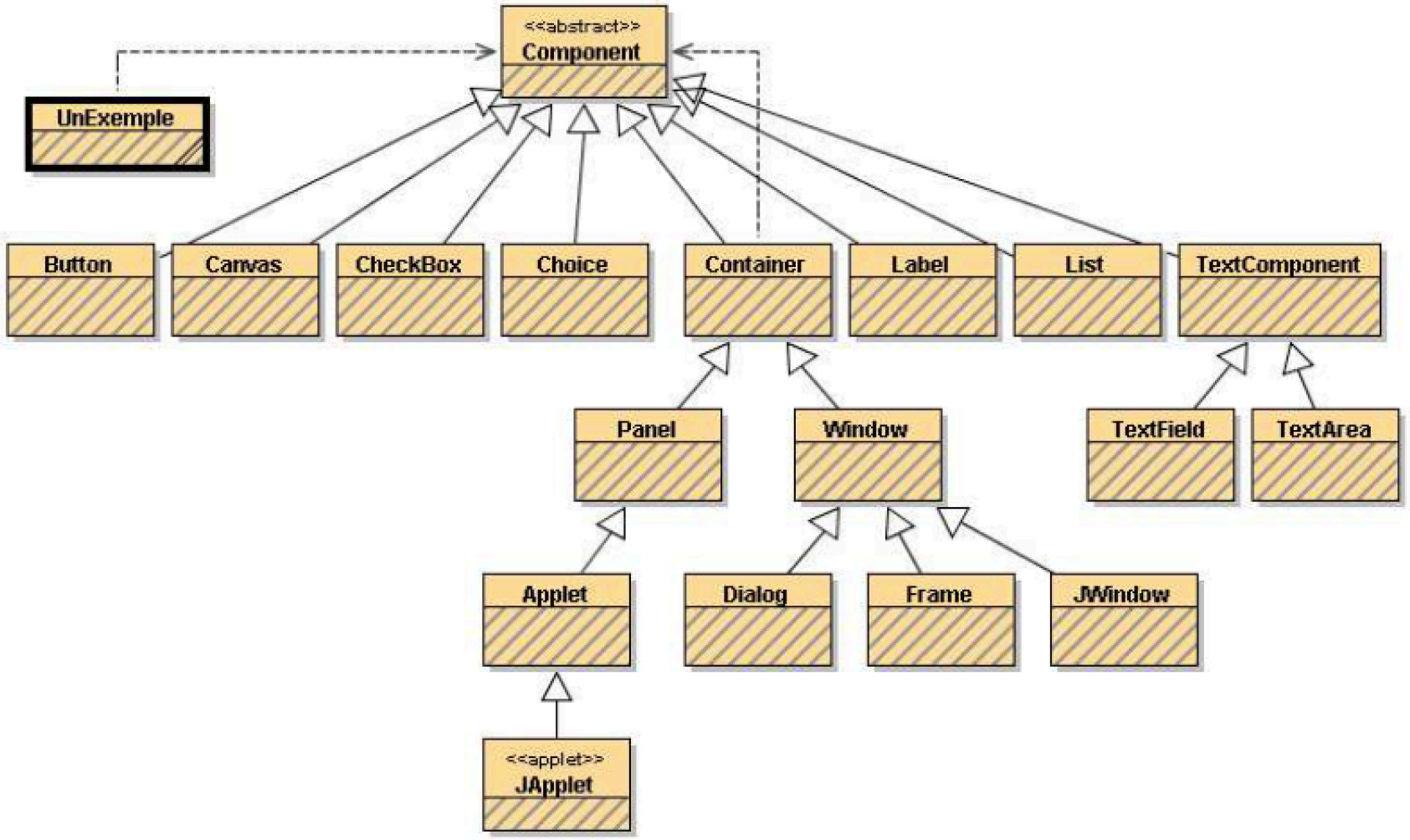
# Composite

- Examples
  - shopping cart: product of a single item or aggregation of multiple items ; how to compute the cost?
  - file system
  - hierarchical state-machine
  - AWT (conteneur graphique)

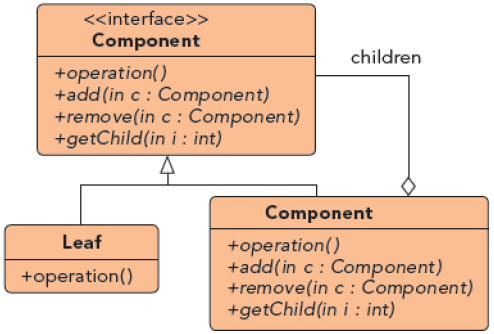


# Composite (example)

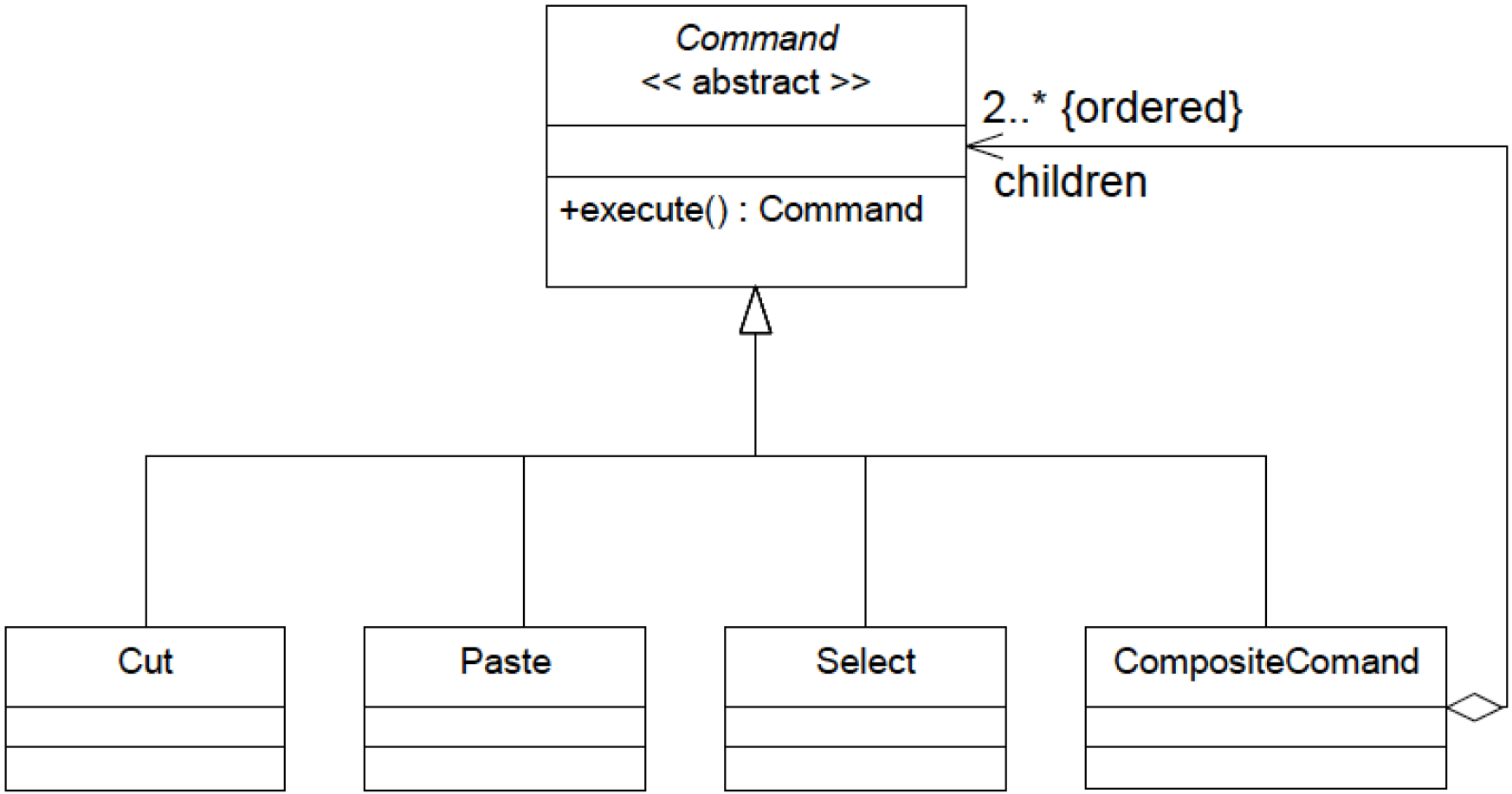


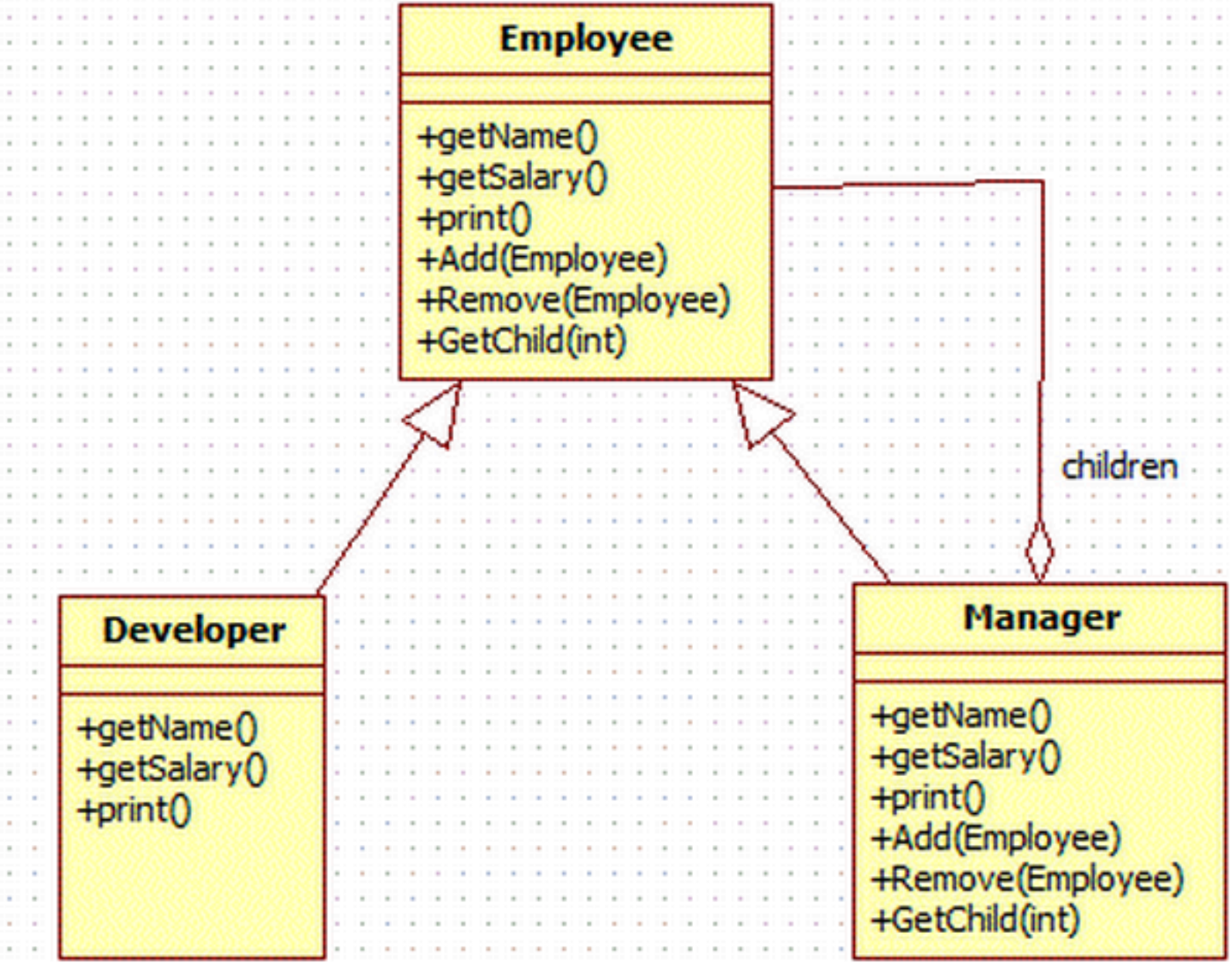

<<abstract>>
Component
UnExemple
Button
Canvas
CheckBox
Choice
Container
Label
List
TextComponent
Panel
Window
TextField
TextArea
Applet
Dialog
Frame
JWindow
<<applet>>
JApplet

# Composite (example)



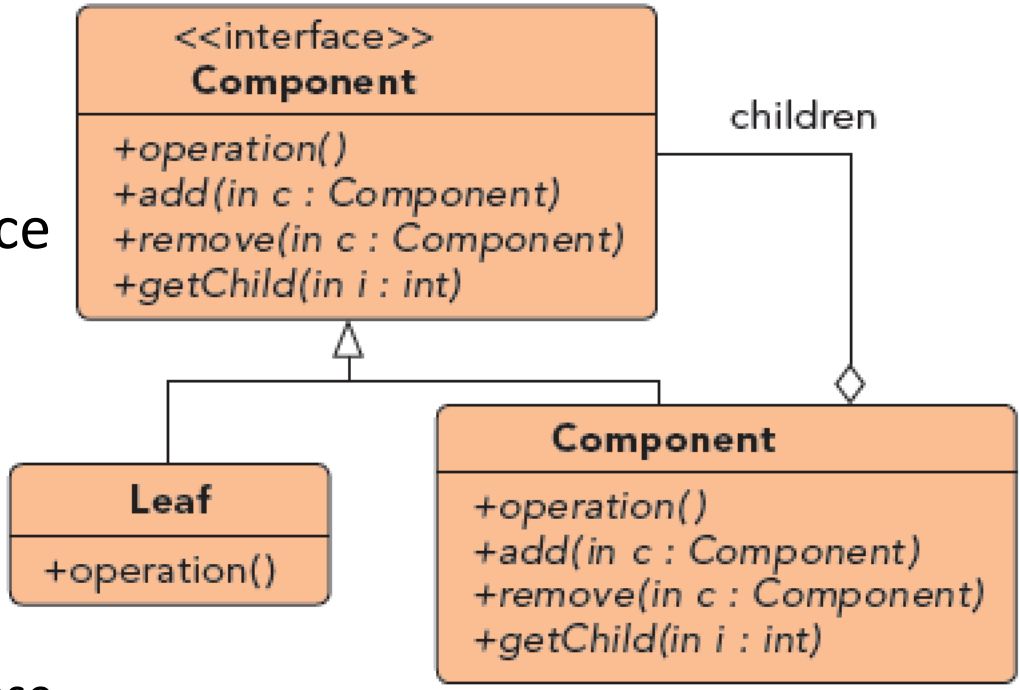

- Component (java.awt.Component)
  - déclare l'interface commune à tous les objets
  - implémente le comportement par défaut pour toutes les classes si nécessaire
  - déclare l'interface pour gérer les composants fils
  - Définit l'interface pour accéder au composant parent (optionnel)
- Leaf représente une feuille (java.awt.Button)
  - Implémentation du comportement
- Composite (java.awt.Container) définit le comportement des composants ayant des fils, stocke les fils et implémente les opérations nécessaires à leur gestion
  - Lien dans la hiérarchie
  - Comportement : fusion des comportements des fils
- Les clients (affichage graphique) utilisent l'interface Component, si le receveur est une feuille la requête est directement traitée, sinon le Composite retransmet habituellement la requête à ses fils en effectuant éventuellement des traitements supplémentaires avant et/ou après


Command
<< abstract >>
+execute() : Command
2..* {ordered}
children
Cut
Paste
Select
CompositeComand


Employee
+getName()
+getSalary()
+print()
+Add(Employee)
+Remove(Employee)
+GetChild(int)
children
Developer
+getName()
+getSalary()
+print()
Manager
+getName()
+getSalary()
+print()
+Add(Employee)
+Remove(Employee)
+GetChild(int)

# Composite

- **Applicability**
  - You want to represent part-whole hierarchies of objects
  - You want to be able to ignore the difference between compositions of objects and individual objects
- **Consequences**
  - Makes the client simple, since it can treat objects and composites uniformly
  - Makes it easy to add new kinds of components
  - Can make the design overly general
    - Operations may not make sense on every class
    - Composites may contain only certain components



# State

# State

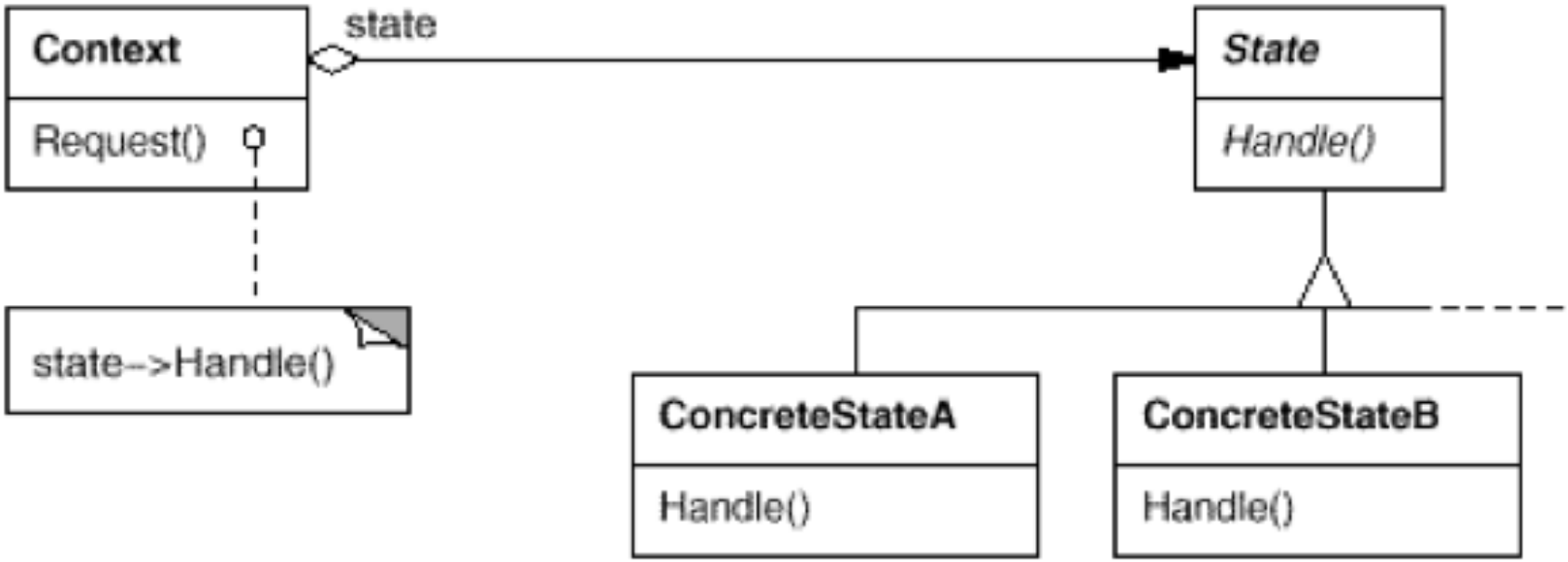
- Exemple
  - Éviter les instructions conditionnelles de grande taille (if then else)
  - Simplifier l'ajout et la suppression d'un état et le comportement qui lui est associé
  - Jeu vidéo: comportement des « bébêtes » avec mémoire de la position, de la vitesse et de la direction (intelligence) => décision de l'évolution de l'état
  - État d'un connexion réseau (recherche, établie, interrompue, fermée)
- Intention
  - Modifier le comportement d'un objet quand son état interne change
  - Obtenir des traitements en fonction de l'état courant
  - Tout est mis en place pour donner l'impression que l'objet lui même a été modifié
- Champs d'application
  - Implanter une partie invariante d'un algorithme
  - Partager des comportements communs d'une hiérarchie de classes

# Patron « State »

- L'objectif est de gérer les états d'un objet par une hiérarchie de classes

- Exemple
  - La fonction "display" d'une icône représentant une connexion change selon l'état.
  - Pour le code qui invoque display, il suffit de changer dynamiquement l'objet qui implémente l'état pour que cette particularité soit insensible

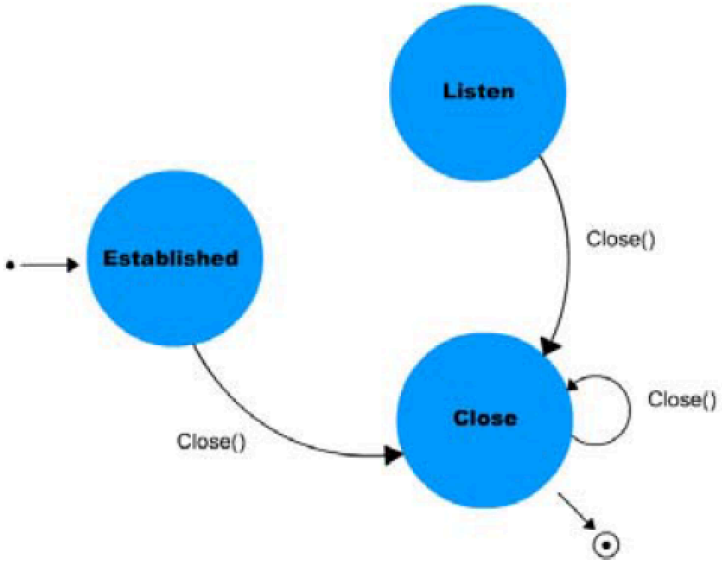
# Patron « State » - structure

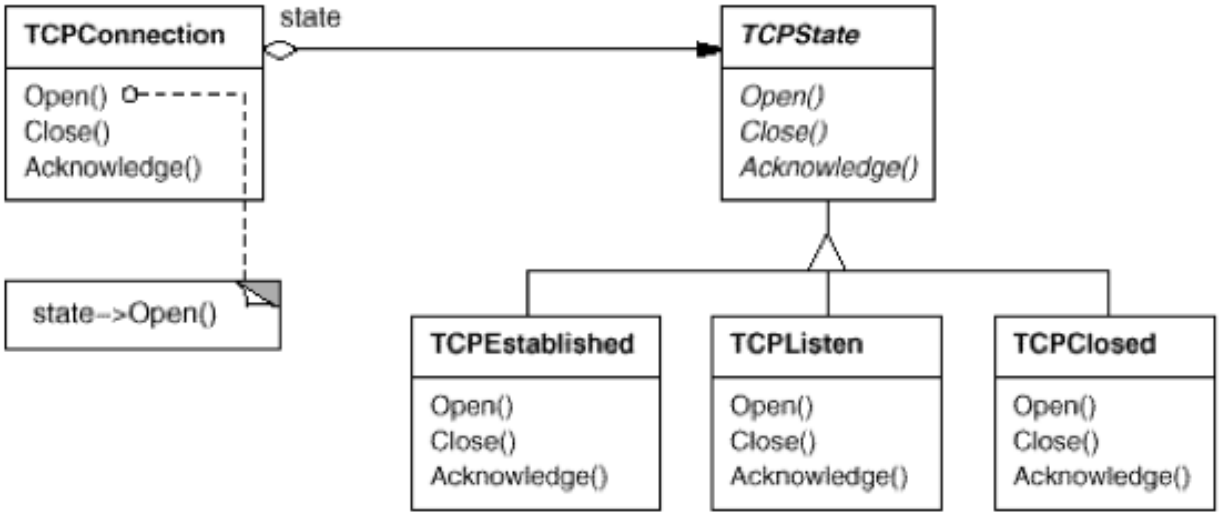



Context est une classe qui permet d'utiliser un objet à état et qui gère une instance d'un objet ConcreteState
State définit une interface qui encapsule le comportement associé avec un état particulier de Context
Les ConcretState implémentent un comportement associé avec l'état de Context

Il revient soit à Context, soit aux ConcreteState de décider de l'état qui succède à un autre état

# Patron « State » - example

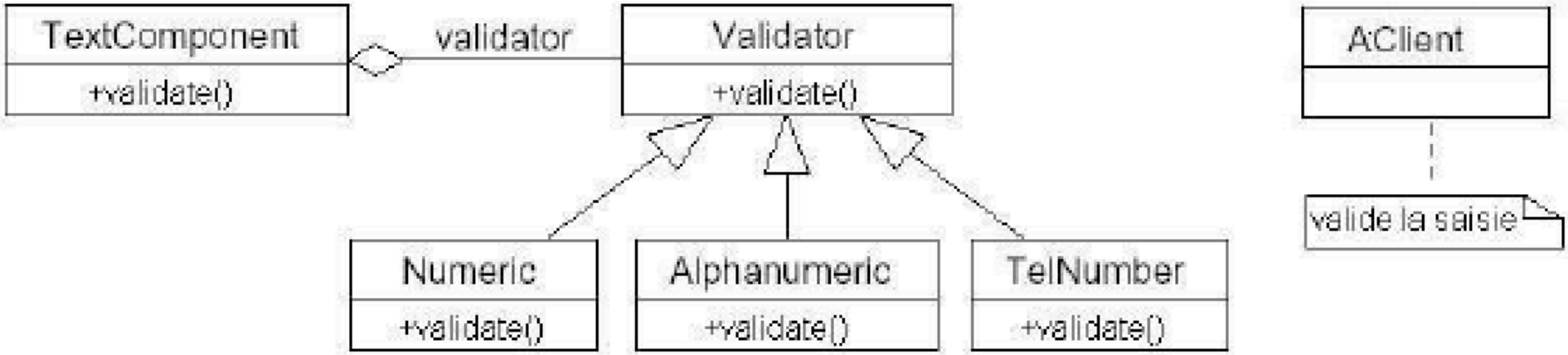


# Strategy

- Plusieurs stratégies de validation selon le type de données : numériques, alphanumériques,...

```
switch(...type){
case NUMERIC : { //… }
case ALPHANUMERIC : { //… }
case TEL_NUMBER : { //…}
}
```

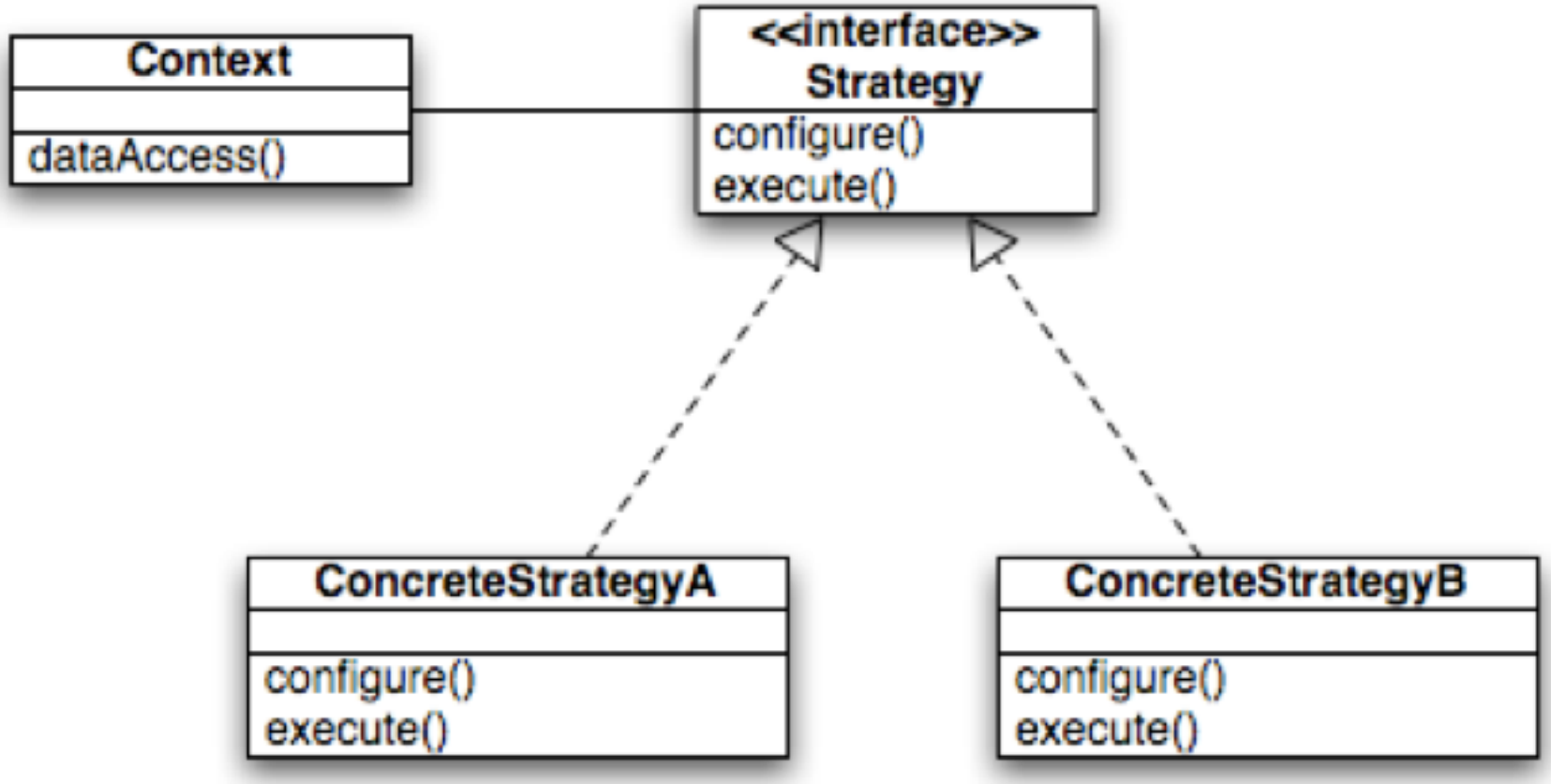
# Patron « Strategy »

- L'objectif est de mettre en œuvre des algorithmes différents avec un choix dynamique de la mise en œuvre
- Déléguer
- Analogie
  - permet de remplacer les « pointeurs de fonction »

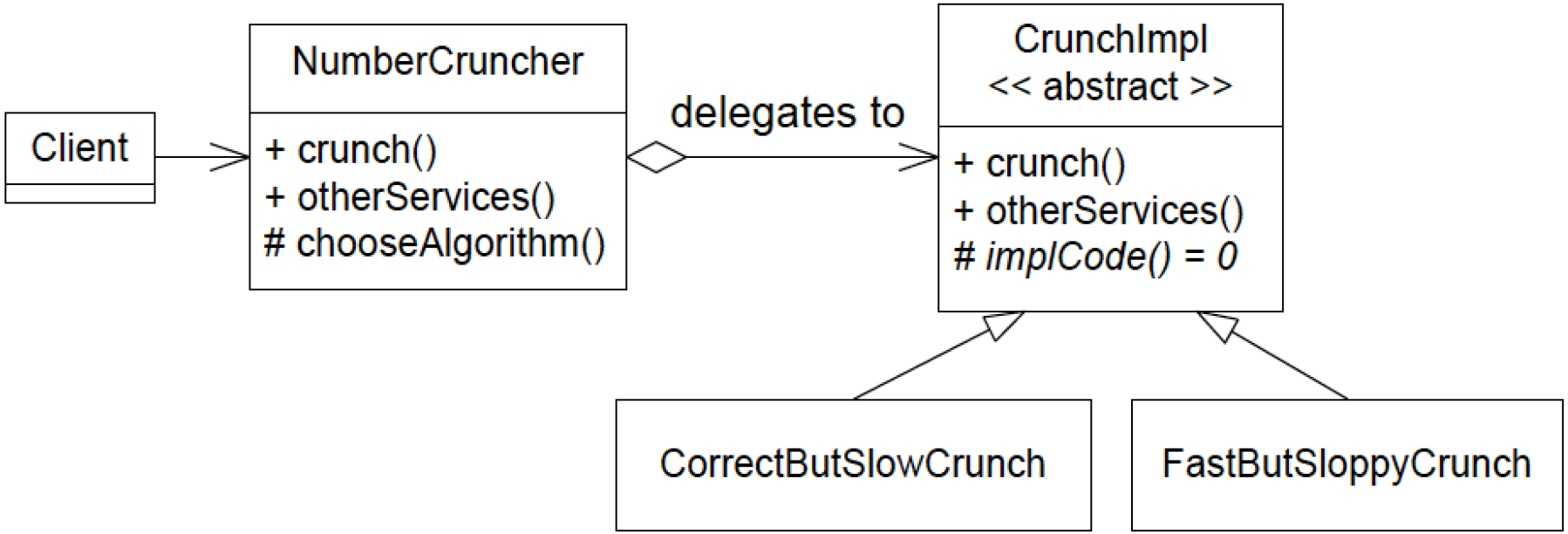
# Patron « Strategy » - rôles

- Strategy
  - définit une interface pour configurer l'algorithme (paramètres) et l'exécuter
- ConcreteStrategy
  - définit une mise en œuvre activée par l'opération d'exécution
- Context
  - désigne l'algorithme concret en vigueur
  - peut contenir des données pour l'algorithme

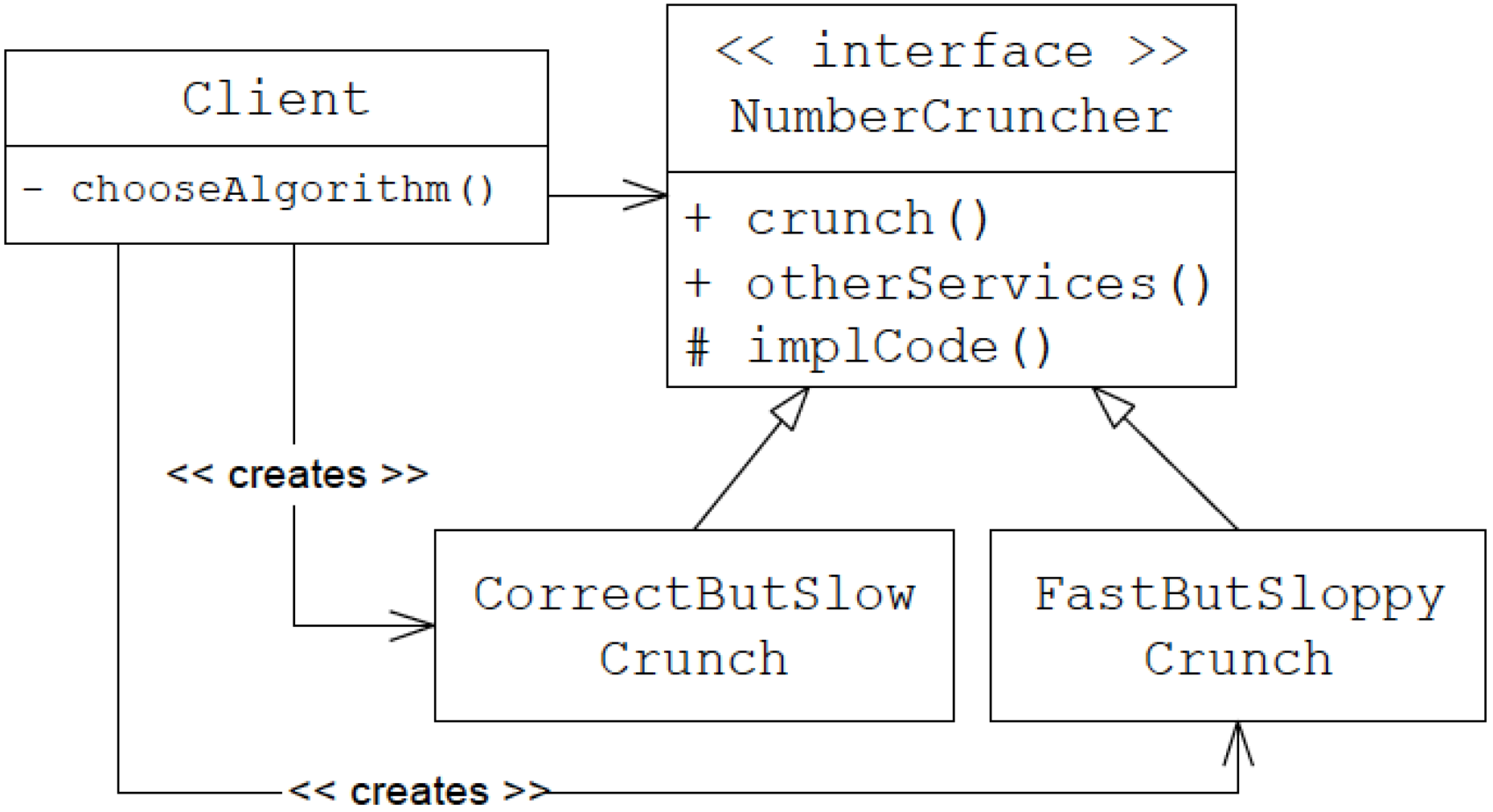
# Patron « Strategy » - structure



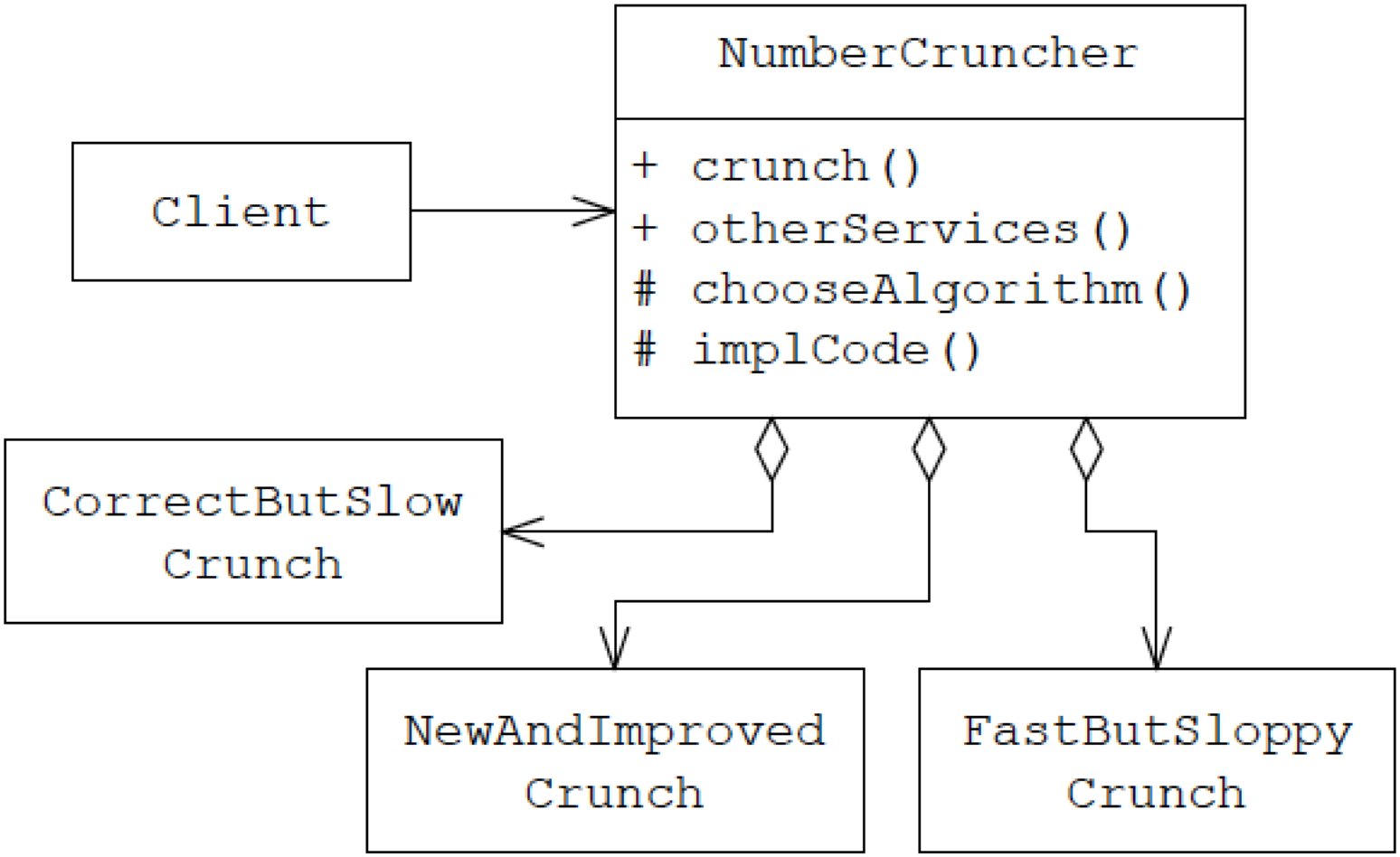
# Machine à calculer (compromis, différentes *stratégies*)



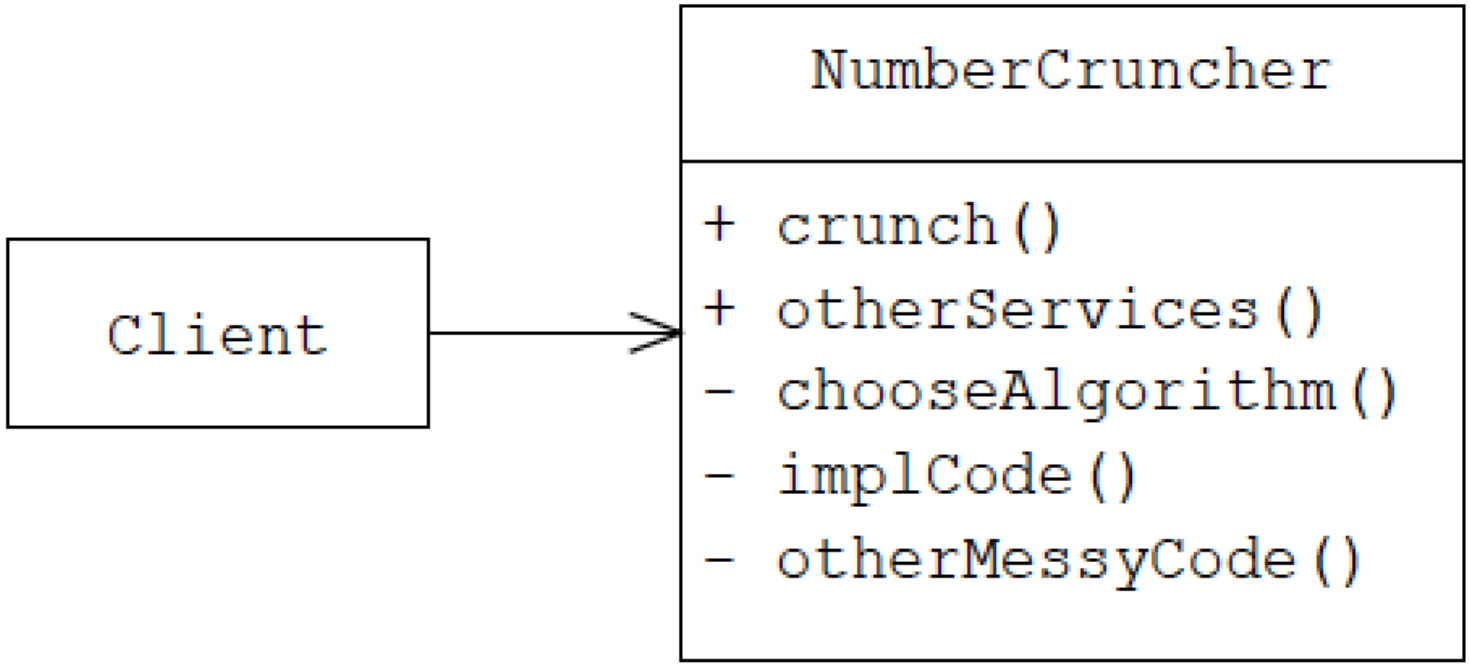
# Bad #1


Client
- chooseAlgorithm()
<< interface >>
NumberCruncher
+ crunch()
+ otherServices()
# implCode()
<< creates >>
CorrectButSlow
Crunch
FastButSloppy
Crunch
<< creates >>

# Bad #2 (if-then-else)

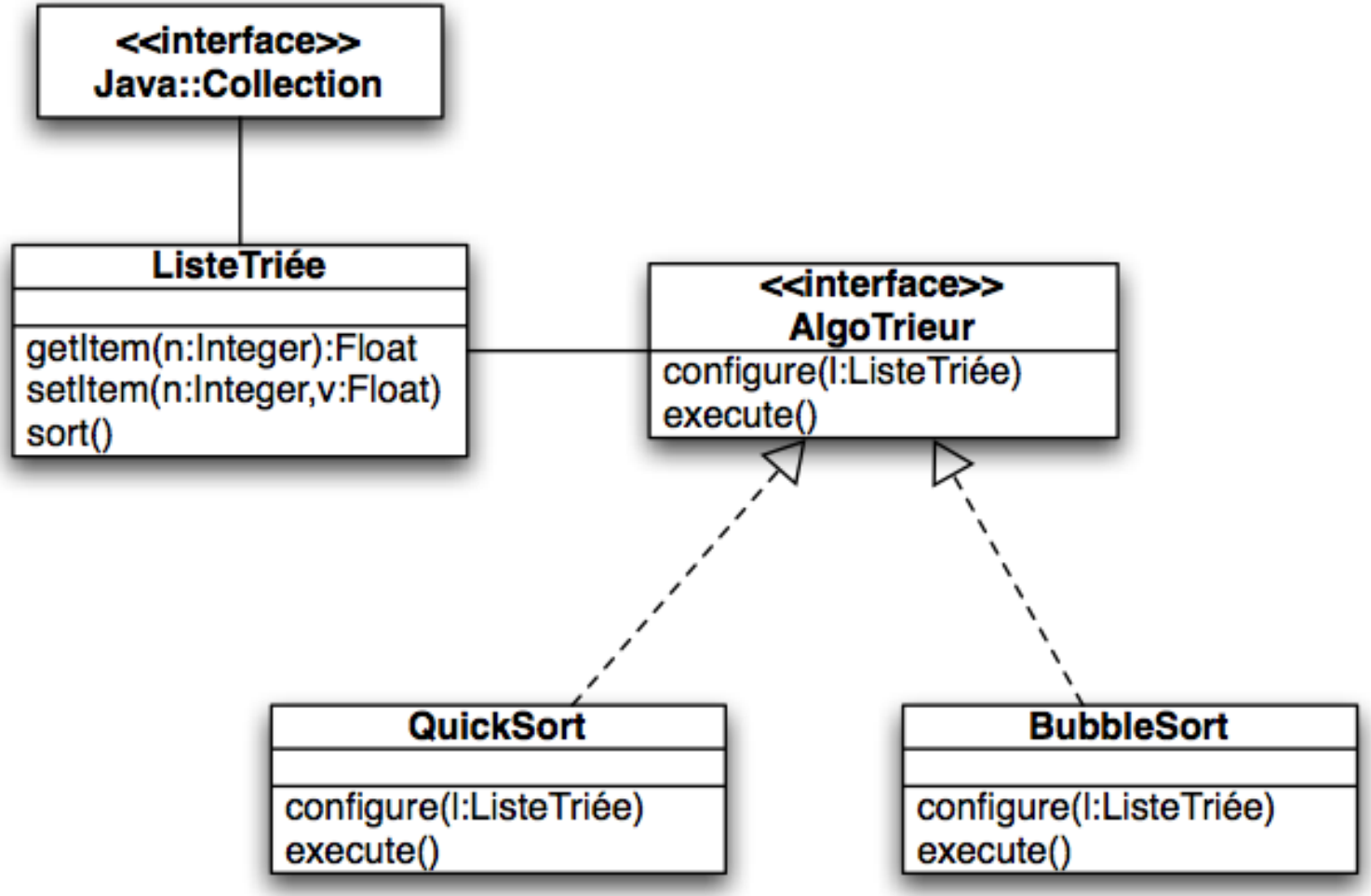


# Bad #3 (everything in one place)



| | |
|---|---|
| + Simplification du code client<br>+ Élimination des boucles conditionnelles<br>+ Extension des algorithmes<br>+ Les clients n'ont pas besoin d'avoir accès au code des classes concrètes<br>+ Les familles d'algorithmes peuvent être rangées hiérarchiquement ou rassemblées dans une super-classe commune | - Le Context ne doit pas changer<br>- Les clients doivent connaître les différentes stratégies<br>- Le nombre d'objets augmente beaucoup<br>- Imaginons une classe Strategy avec beaucoup de méthodes. Chaque sous-classe connaît ses méthodes mais peut être qu'elle ne les utilisera pas |

# Patron « Strategy » - example

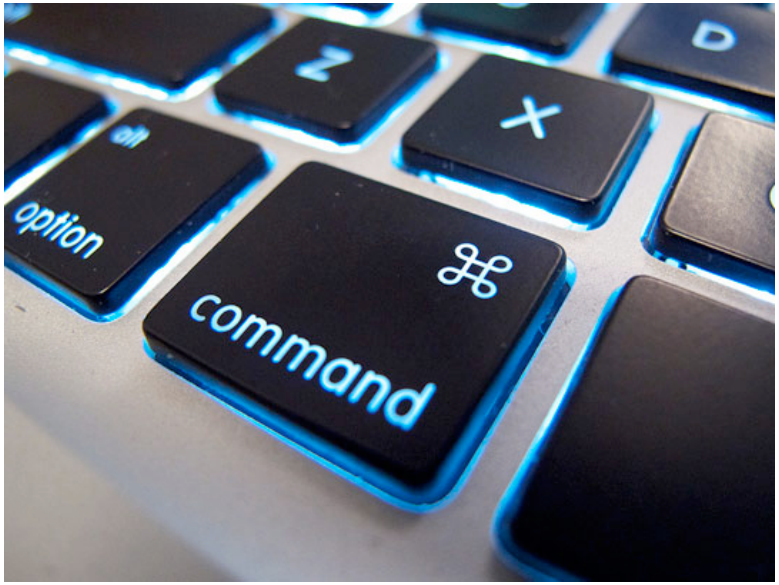


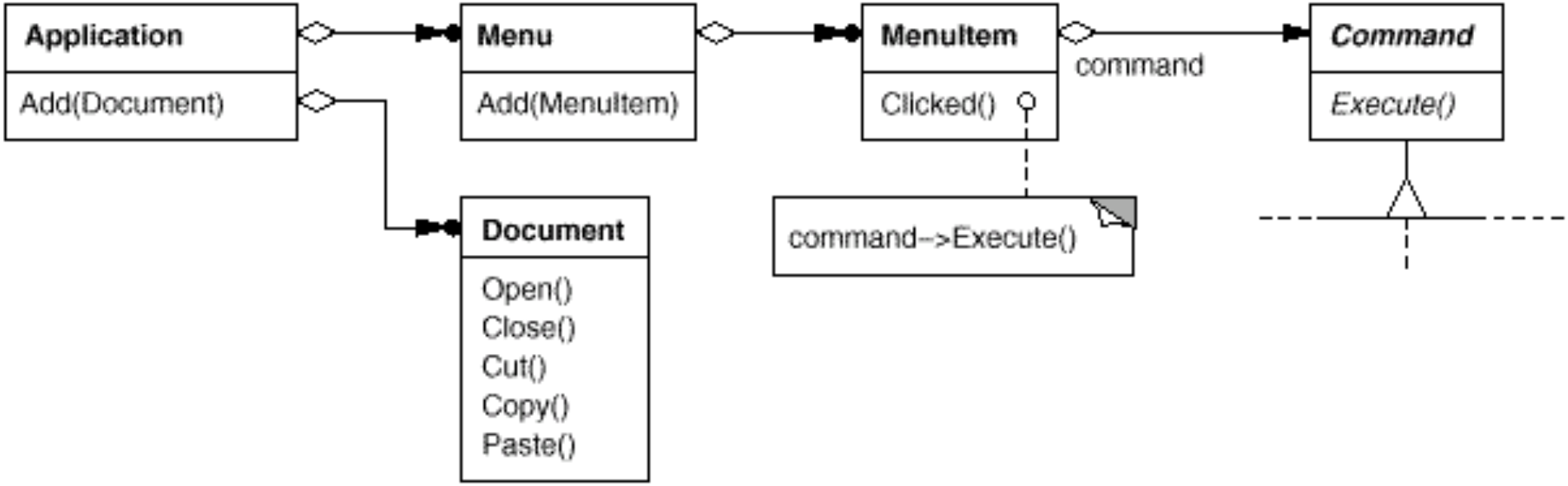
# Command

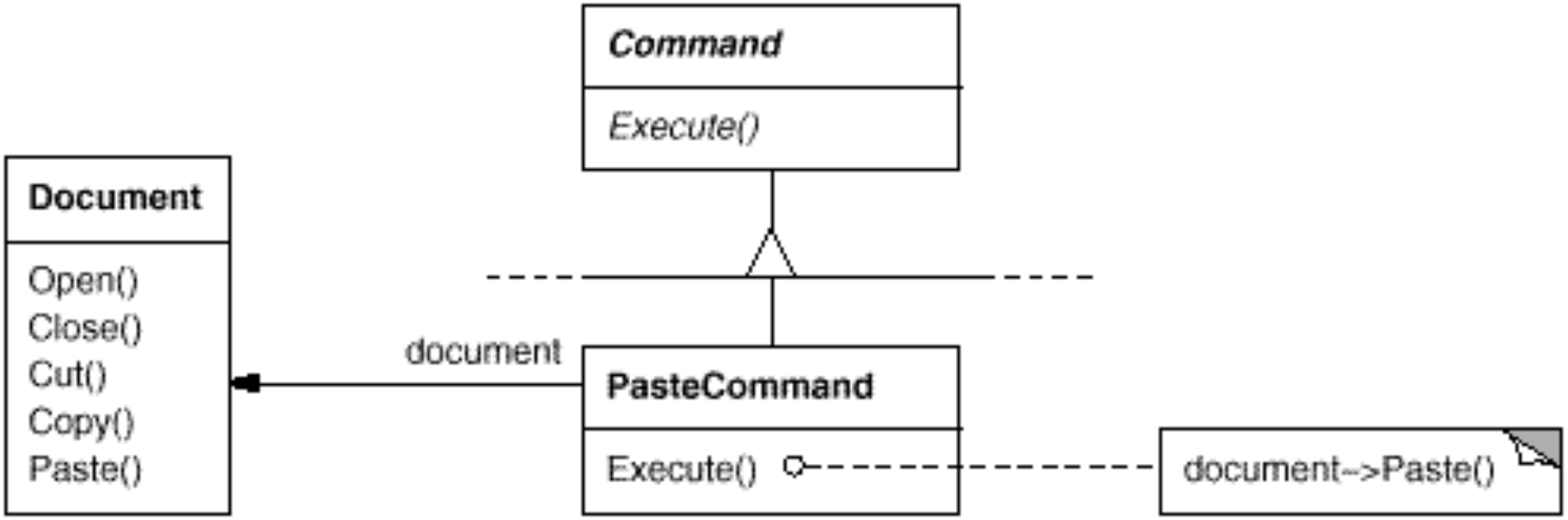
# Patron « Command »

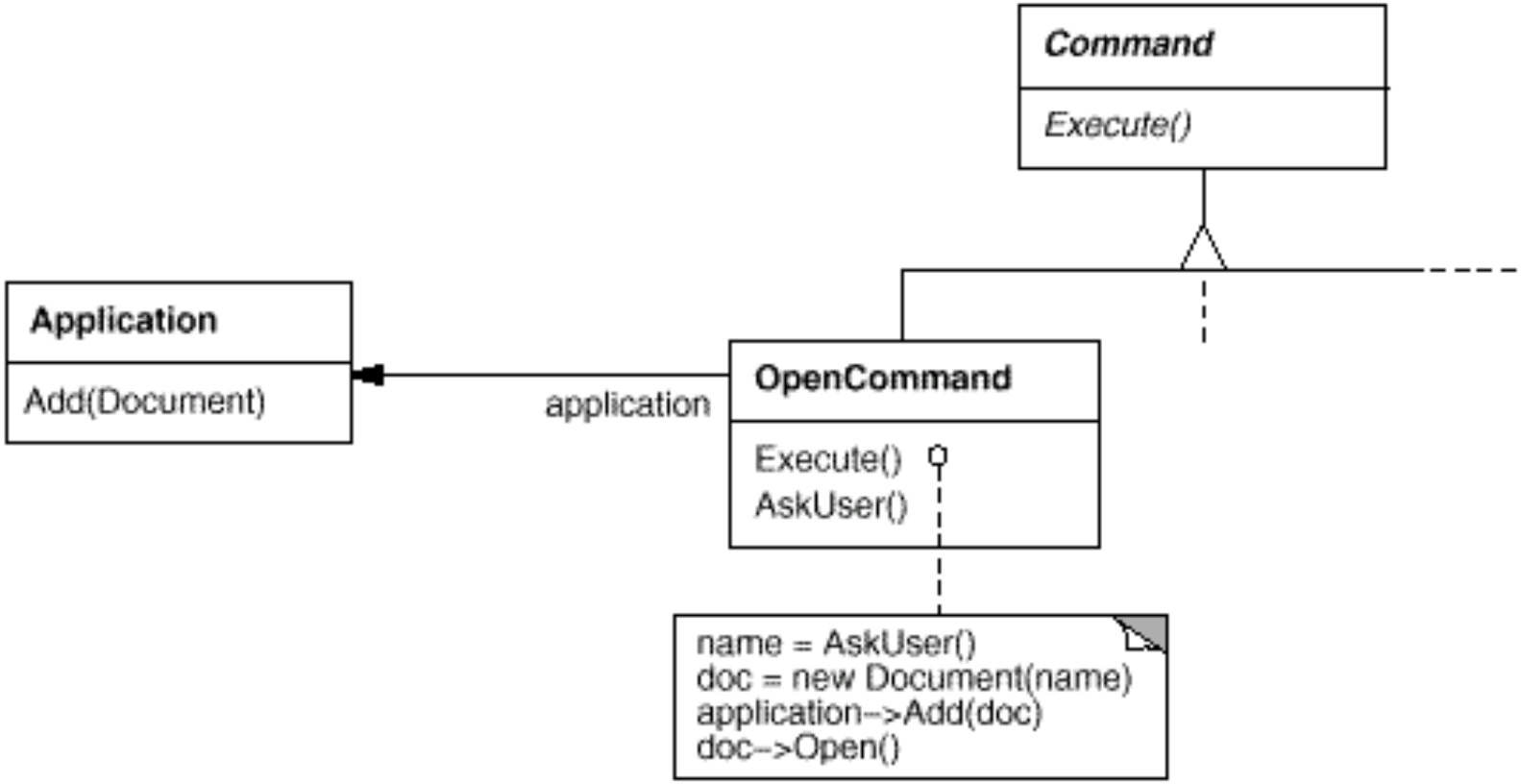
- L'objectif est de découpler
  - le choix d'une action à faire dans une certaine situation (e.g., undo)
  - la détection de la situation et l'exécution de l'action décidée

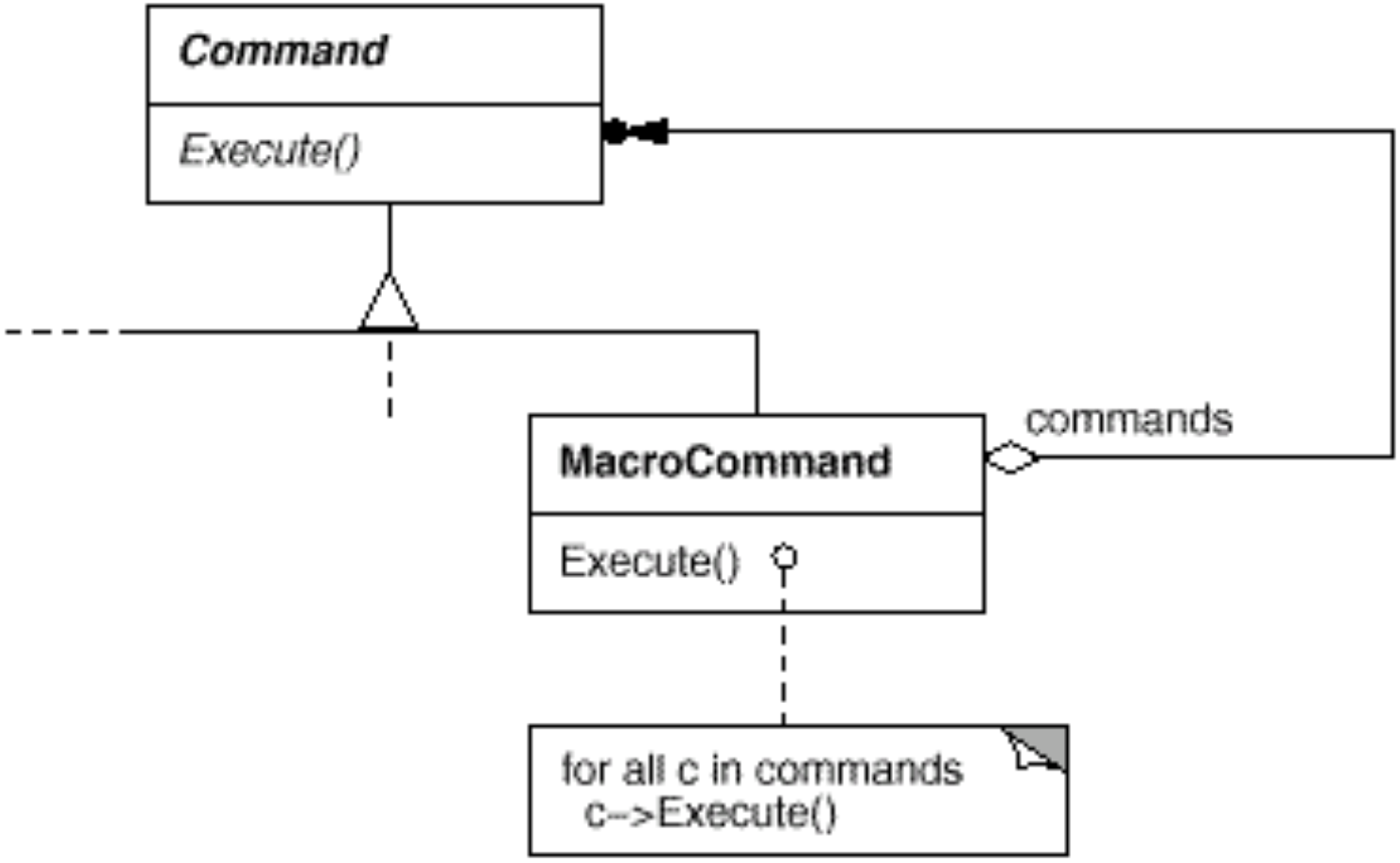
Encapsuler une requête comme un objet




Application
Add(Document)
Menu
Add(MenuItem)
MenuItem
Clicked()
command
Command
Execute()
Document
Open()
Close()
Cut()
Copy()
Paste()
command->Execute()


Command
Execute()
Document
Open()
Close()
Cut()
Copy()
Paste()
document
PasteCommand
Execute()
document->Paste()

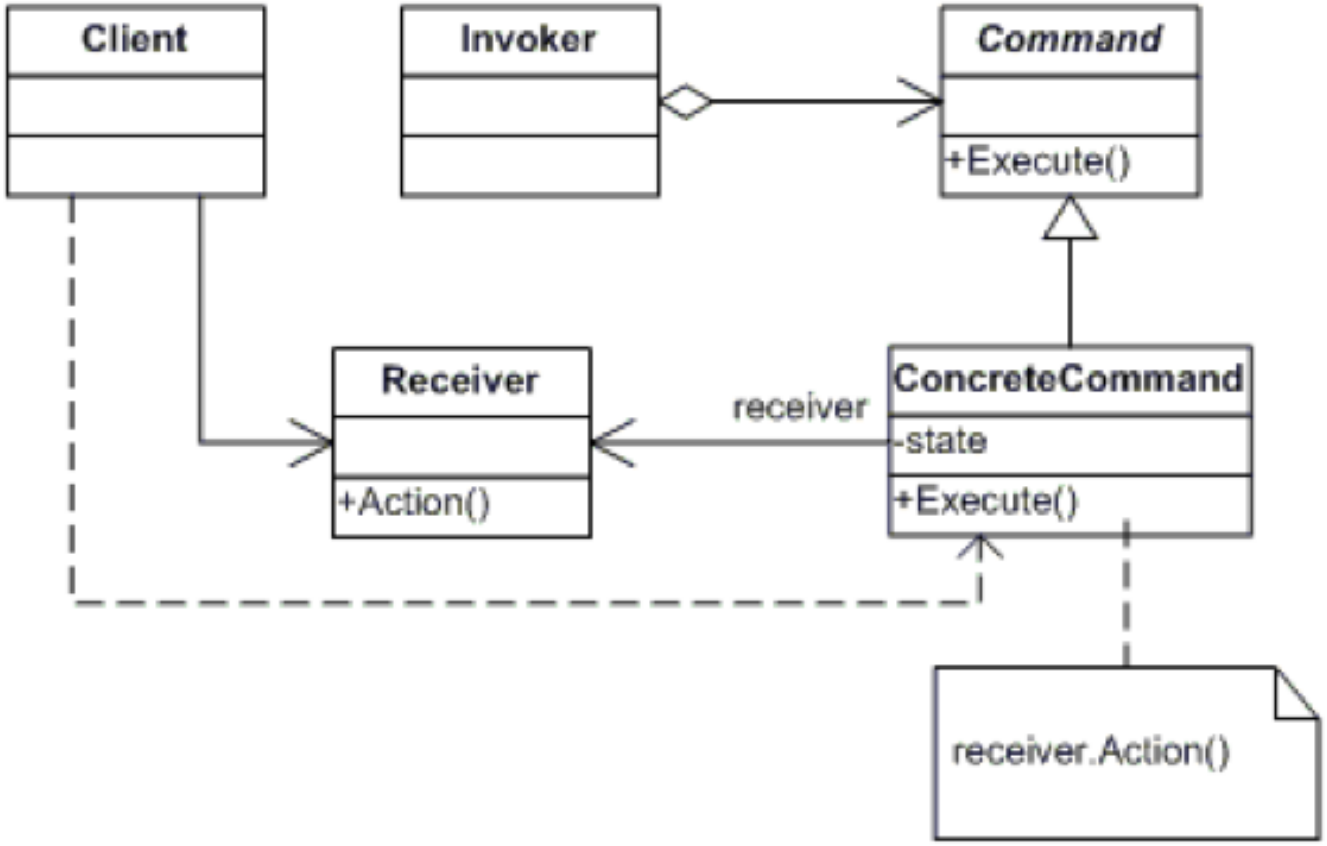

Command
Execute()
Application
Add(Document)
application
OpenCommand
Execute()
AskUser()
name = AskUser()
doc = new Document(name)
application->Add(doc)
doc->Open()


Command
Execute()
MacroCommand
Execute()
commands
for all c in commands
c->Execute()

# Patron « Command » - rôles

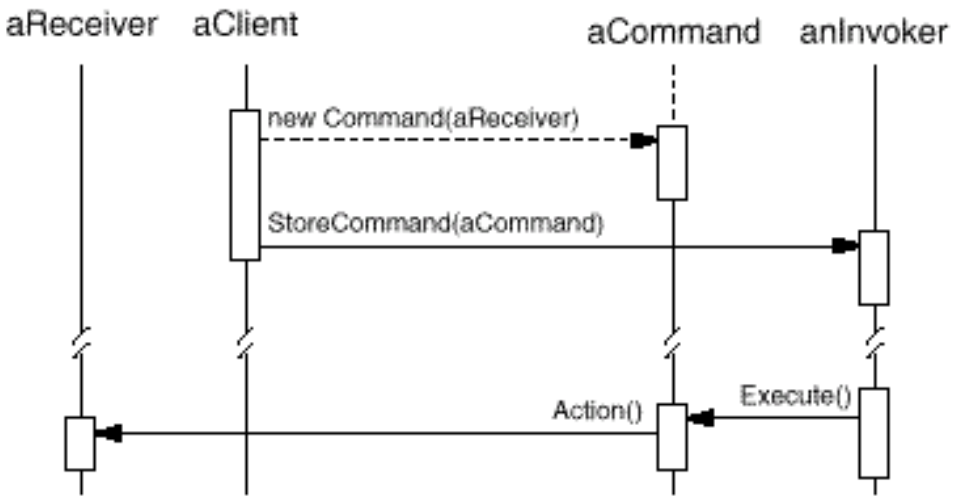
- client
  - chargé de la création des commandes concrètes et de leur association avec des situations
- invoker
  - chargé de détecter une situation et de faire exécuter la commande correspondante
- receiver
  - effectue le travail requis par la commande
- command
  - déclare une opération d'exécution
- concrete command
  - fournit une méthode pour l'opération d'exécution

# Patron « Command » - structure

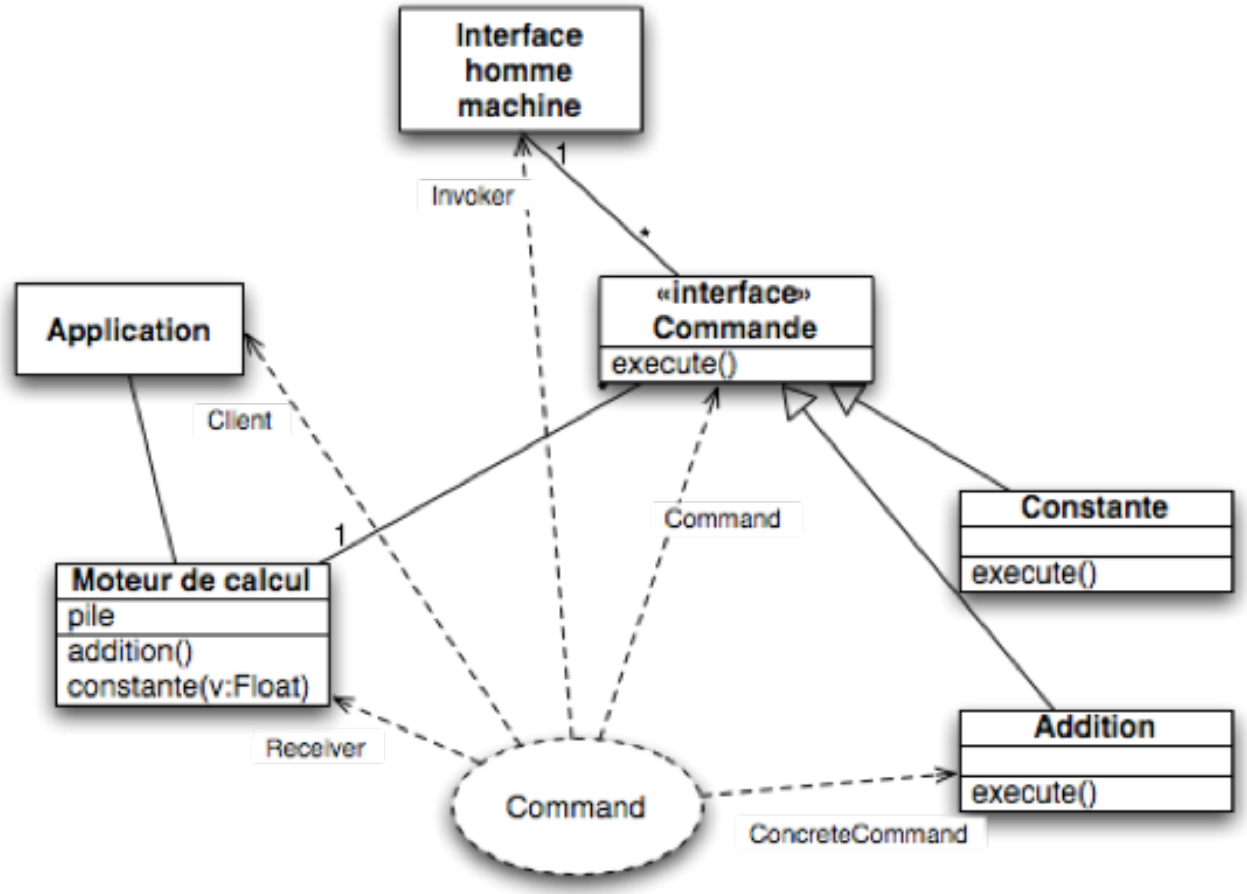


# Collaborations

- Le client crée l'objet **`ConcreteCommand`** et spécifie le destinataire.
- L'objet **`Invoker`** emmagasine l'objet **`ConcreteCommand`**.
- L'objet **`Invoker`** émet une requête en invoquant "**`execute`**" sur la commande.
  - Lorsque la commande est réversible, **`ConcreteCommand`** emmagasine l'état nécessaire pour revenir dans l'état précédant l'invocation de "**`execute`**".
- L'objet **`ConcreteCommand`** invoque les opérations du destinataire pour exécuter la requête.



# Patron « Command » - example



# Intention

- Encapsuler une requête sous forme d'objet
  - paramétrer les clients avec différentes requêtes,
  - files de requêtes
  - "logs" de requêtes
  - support d'opérations réversibles (« undo »)

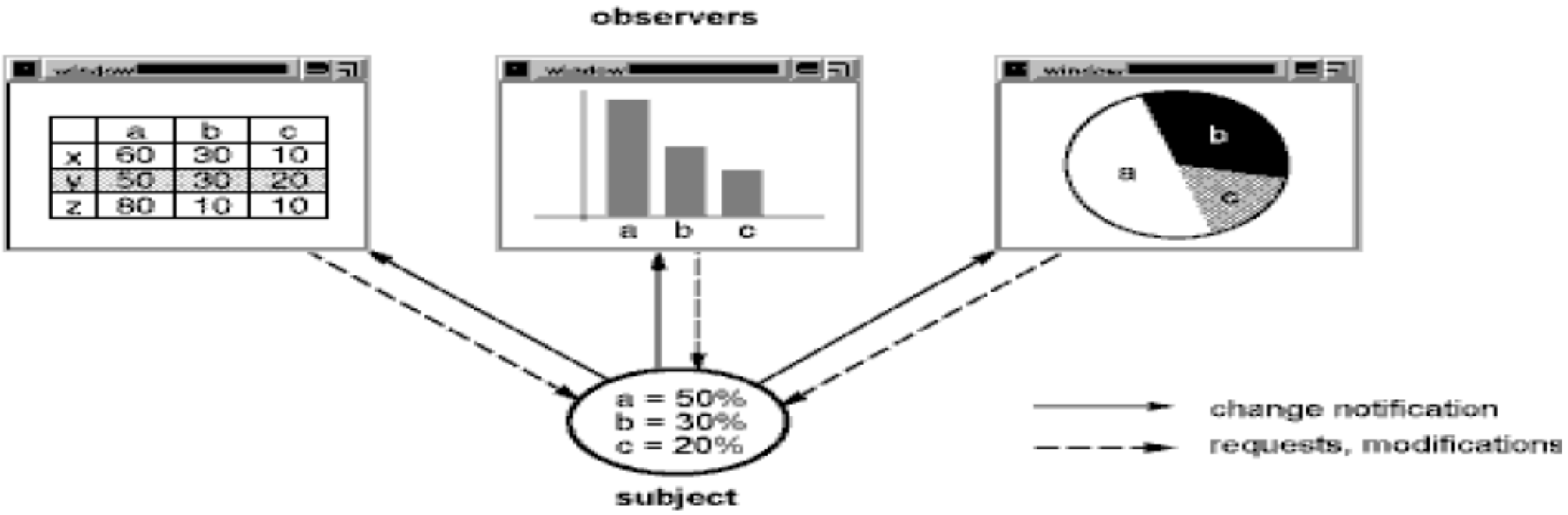
# Quand appliquer le patron Command?

- Pour paramétriser des objets par une action à exécuter
  - Exemple: objets MenuItem
  - Les commandes servent à remplacer les "callback" dans les langages orientés objets.
    - Dans un langage procédural >>> une fonction **callback** function, une fonction enregistrée en quelque part qui doit être appelée plus tard

- Pour spécifier, mettre en file, et exécuter les requêtes à des moments différents
  - Le cycle de vie d'un objet Command est indépendant de la requête originale

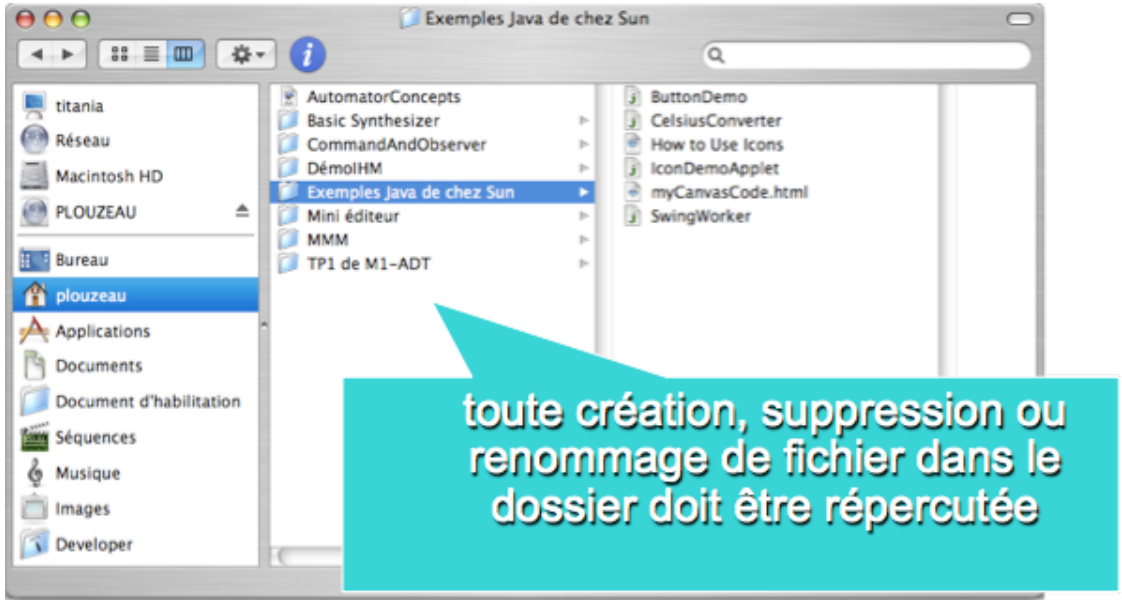
- Pour implémenter des opérations réversibles.
  - L'exécution de la commande peut conserver l'état dans la commande elle-même pour inverser son effet.
  - Ajout d'une méthode "unexecute"
  - Les commandes exécutées sont placées dans une liste.

# Observer


observers
window
a b c
x 60 30 10
y 50 30 20
z 80 10 10
window
a b c
window
a
b
c
a = 50%
b = 30%
c = 20%
subject
change notification
requests, modifications

# Patron « Observer »

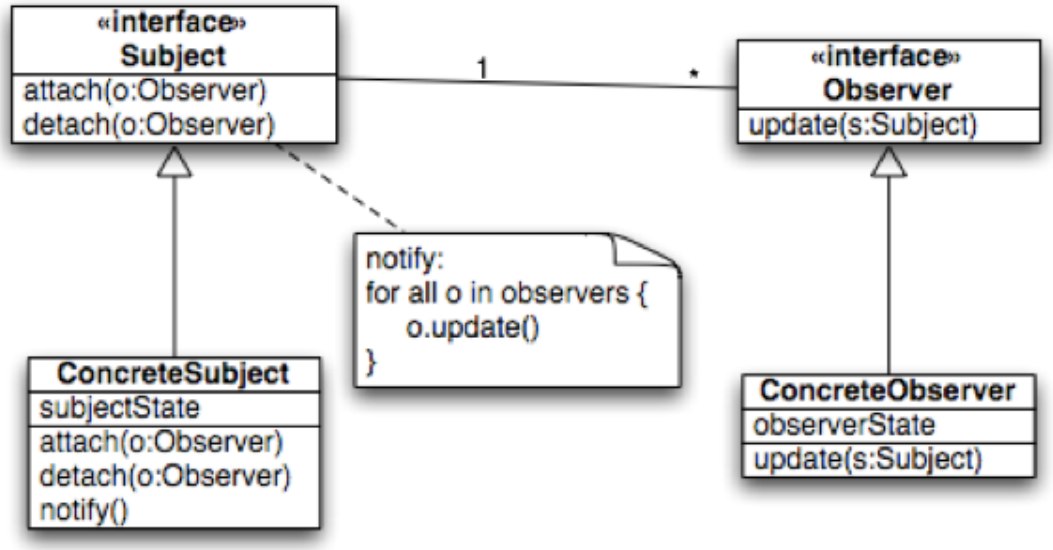
- L'objectif est de propager les changements d'état d'un objet vers d'autres objets



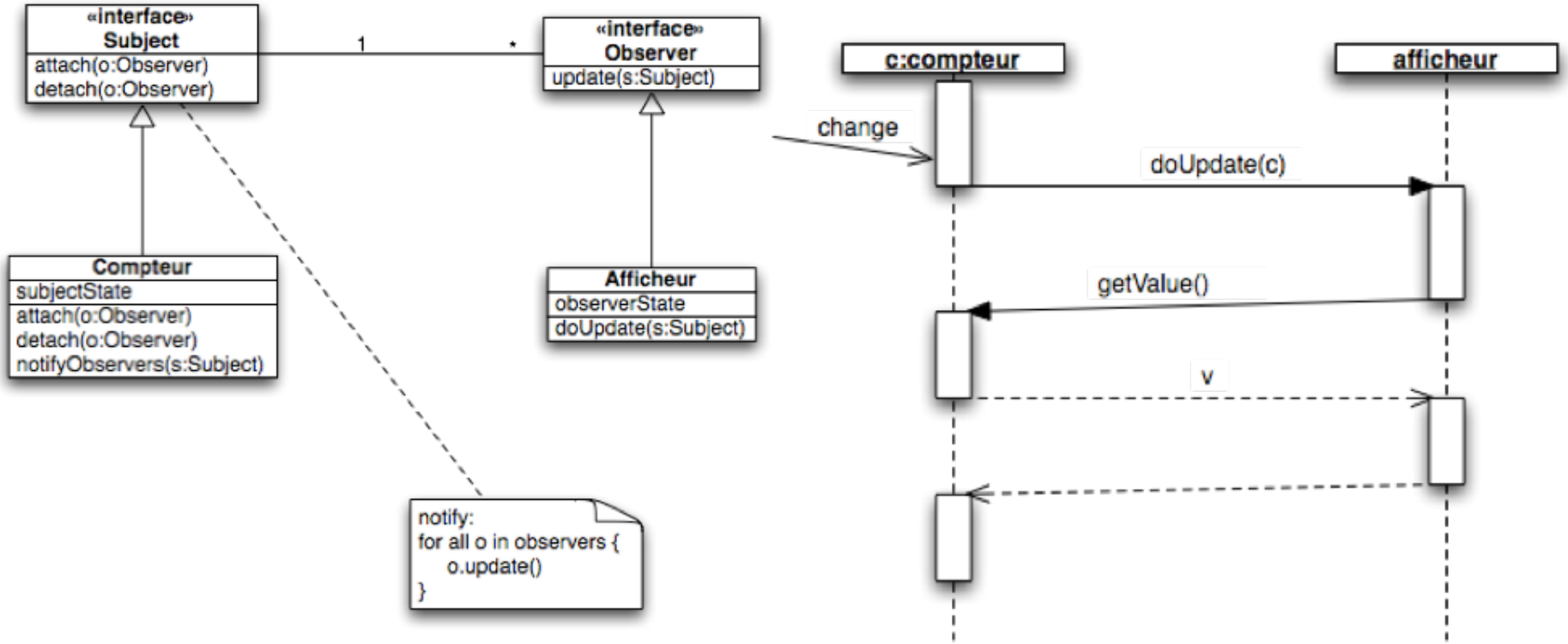

# Patron « Observer » - rôles

- subject
  - Comporte un état interne
    - type non spécifié
    - un patron de conception est indépendant de ce genre de détail
  - Est chargé de gérer une collection d'abonnés capable de recevoir des notifications
  - Est chargé d'envoyer un message aux abonnés lorsque son état change
- observer
  - Est capable de réagir à la réception d'un message de notification venant d'un sujet

# Patron « Observer » - structure



# Patron « Observer » - example



UNIVERSITÉ DE RENNES 1

ESIR ECOLE SUPERIEURE D'INGENIEURS DE RENNES

# Méthodes de Développement Industriel (MDI)

**Mathieu Acher**

http://www.mathieuacher.com

*Associate Professor*

*University of Rennes 1*

UMR IRISA

Inria informatics mathematics

# Objectifs de MDI

- Méthodes de développement industriel (MDI)
  - En fait: génie logiciel / software engineering
  - Comment développer des systèmes logiciels de plus en plus complexe?
- #1 Prendre conscience de la complexité des systèmes logiciels actuels et à venir
  - Les enjeux et l'impact sur le métier
- #2 Modélisation
  - UML, SysML
- **#3 Design patterns, refactoring, test**
  - **OO avancé**
- #4 Méthodes

# Previously

- Composite
- State
- Strategy
- Command
- Observer

# Template Method

```
public class PlainTextDocument {

   ...

   public void printPage (Page page) {

      printPlainTextHeader();    // Unique to PlainTextDocument
      System.out.println(page.body());
      printPlainTextFooter();    // Unique to PlainTextDocument

   }

   ...

}
```

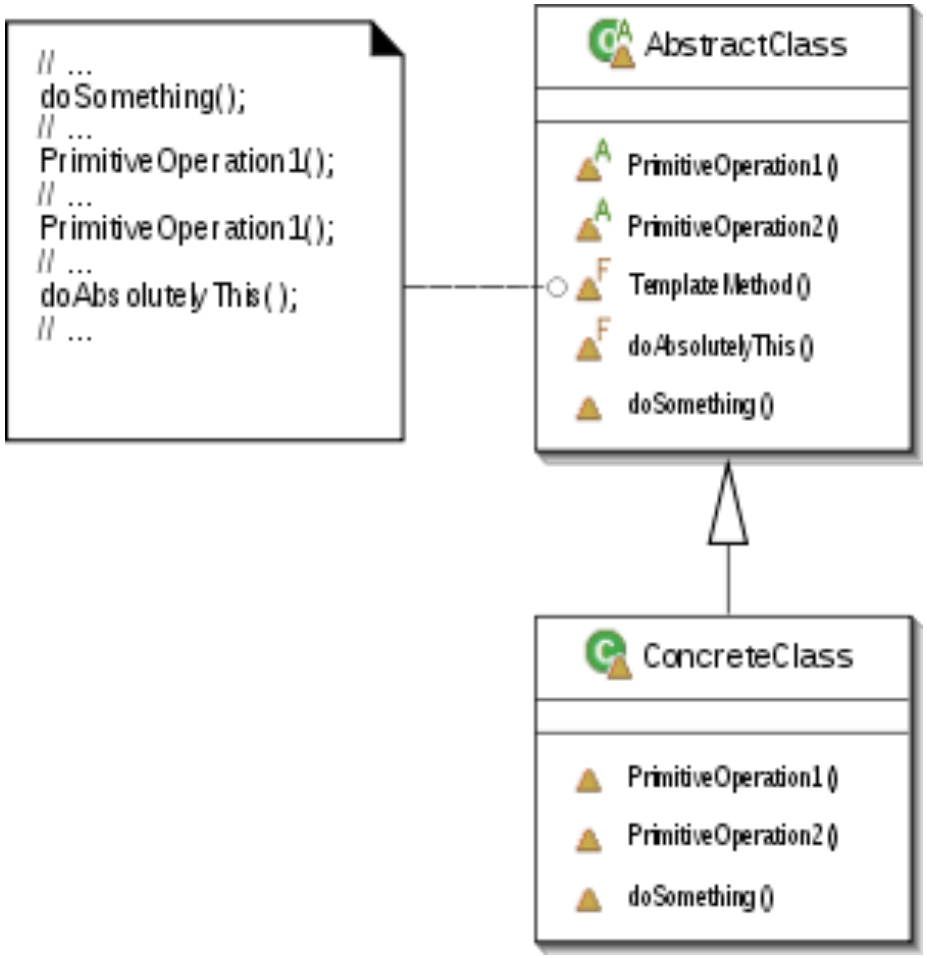
```
public class HtmlTextDocument {

   ...

   public void printPage (Page page) {

      printHtmlTextHeader();      // Unique to HtmlTextDocument
      System.out.println(page.body());
      printHtmlTextFooter();      // Unique to HtmlTextDocument

   }

   ...

}
```

```
public abstract class TextDocument {
  ...
  public final void printPage (Page page) {
    printTextHeader();
    printTextBody(page);
    printTextFooter();
  }
  public abstract void printTextHeader();
  public final void printTextBody(Page page) {
    System.out.println(page.body());
  }
  public abstract void printTextFooter();
  ...
}
```

```
public class PlainTextDocument extends TextDocument {
   ...
   public void printTextHeader () {
      // Code for header plain text header here.
   }

   public void printTextFooter () {
      // Code for header plain text footer here.
   }
   ...
}
```

# Template

- Applicability
  - When an algorithm consists of **varying and invariant parts** that must be customized
  - When **common behavior** in subclasses should be factored and localized to avoid code duplication
  - To control subclass extensions to specific operations
- Consequences
  - Code reuse
  - Inverted “Hollywood” control: don’t call us, we’ll call you
  - Ensures the invariant parts of the algorithm are not changed by subclasses



```
public class Manufacturing {
   ...
   // A template method!
   public final void makePart () {
      operation1();
      operation2();
   }

   public void operation1() {
      // Default behavior for Operation 1
   }

   public void operation2() {
      // Default behavior for Operation 2
   }
   ...
```

```
public class MyManufacturing {
   ...
   // We want to do behavior between operation1() and
   // operation2() of makePart(), so we override operation2()
   // as follows.  (Note: we could just as easily have
   // overridden operation1().)
   public void operation2() {
     // Put behavior we want to do BEFORE the normal Operation2
     //  here!
     super.operation2();
   }
   ...
}
```

```
public class Manufacturing {
   ...
   // A template method!
   public final void makePart () {
      operation1();
      hook();  // A hook method
      operation2();
   }
   // Do nothing hook method.
   public void hook() {}
   ...
}
```

# Template Method in an existing API

- http://developer.classpath.org/doc/javax/swing/table/
- TableModel
- AbstractTableModel
- DefaultTableModel

- Demo

```
38:
39: package javax.swing.table;
40:
41: import java.io.Serializable;
42: import java.util.EventListener;
43:
44: import javax.swing.event.EventListenerList;
45: import javax.swing.event.TableModelEvent;
46: import javax.swing.event.TableModelListener;
47:
48: /**
49:  * A base class that can be used to create implementations of the
50:  * {@link TableModel} interface.
51:  *
52:  * @author Andrew Selkirk
53:  */
54: public abstract class AbstractTableModel implements TableModel, Serializable
55: {
```

---

`public interface` **`TableModel`**

---

A `TableModel` is a two dimensional data structure that can store arbitrary `Object` instances, usually for the purp with it.

The `DefaultTableModel` class provides one implementation of this interface.

## Method Summary

| | |
|---|---|
| `void` | `addTableModelListener(TableModelListener listener)`<br>Adds a listener to the model. |
| `Class` | `getColumnClass(int columnIndex)`<br>Returns the `Class` for all `Object` instances in the specified column. |
| `int` | `getColumnCount()`<br>Returns the number of columns in the model. |
| `String` | `getColumnName(int columnIndex)`<br>Returns the name of a column in the model. |
| `int` | `getRowCount()`<br>Returns the number of rows in the model. |
| `Object` | `getValueAt(int rowIndex, int columnIndex)`<br>Returns the value (`Object`) at a particular cell in the table. |
| `boolean` | `isCellEditable(int rowIndex, int columnIndex)`<br>Returns `true` if the cell is editable, and `false` otherwise. |
| `void` | `removeTableModelListener(TableModelListener listener)`<br>Removes a listener from the model. |
| `void` | `setValueAt(Object aValue, int rowIndex, int columnIndex)`<br>Sets the value at a particular cell in the table. |

```
38:
39: package javax.swing.table;
40:
41: import java.io.Serializable;
42: import java.util.EventListener;
43:
44: import javax.swing.event.EventListenerList;
45: import javax.swing.event.TableModelEvent;
46: import javax.swing.event.TableModelListener;
47:
48: /**
49:  * A base class that can be used to create implementations of the
50:  * {@link TableModel} interface.
51:  *
52:  * @author Andrew Selkirk
53:  */
54: public abstract class AbstractTableModel implements TableModel, Serializable
55: {
```

```
92:   /**
93:    * Return the index of the specified column, or <code>-1</code> if there is
94:    * no column with the specified name.
95:    *
96:    * @param columnName  the name of the column (<code>null</code> not permitted).
97:    *
98:    * @return The index of the column, -1 if not found.
99:    *
100:   * @see #getColumnName(int)
101:   * @throws NullPointerException if <code>columnName</code> is
102:   *         <code>null</code>.
103:   */
104:  public int findColumn(String columnName)
105:  {
106:    int count = getColumnCount();
107:
108:    for (int index = 0; index < count; index++)
109:      {
110:        String name = getColumnName(index);
111:
112:        if (columnName.equals(name))
113:          return index;
114:    }
115:
116:    // Unable to locate.
117:    return -1;
118:  }
119:
```

developer.classpath.org/doc/javax/swing/table/AbstractTableModel.html

## Constructor Summary

| | |
|---|---|
| AbstractTableModel() | Creates a default instance. |

## Method Summary

| | |
|---|---|
| void | addTableModelListener(TableModelListener listener)<br>Adds a listener to the table model. |
| | extends EventListener> T[] getListeners(Class listenerType)<br>Returns an array of listeners of the given type that are registered with this model. |
| int | findColumn(String columnName)<br>Return the index of the specified column, or -1 if there is no column with the specified name. |
| void | fireTableCellUpdated(int row, int column)<br>Sends a TableModelEvent to all registered listeners to inform them that a single cell has been updated. |
| void | fireTableChanged(TableModelEvent event)<br>Sends the specified event to all registered listeners. |
| void | fireTableDataChanged()<br>Sends a TableModelEvent to all registered listeners to inform them that the table data has chan |
| void | fireTableRowsDeleted(int firstRow, int lastRow)<br>Sends a TableModelEvent to all registered listeners to inform them that some rows have been |
| void | fireTableRowsInserted(int firstRow, int lastRow)<br>Sends a TableModelEvent to all registered listeners to inform them that some rows have been |
| void | fireTableRowsUpdated(int firstRow, int lastRow)<br>Sends a TableModelEvent to all registered listeners to inform them that some rows have been |
| void | fireTableStructureChanged()<br>Sends a TableModelEvent to all registered listeners to inform them that the table structure has |
| Class | getColumnClass(int columnIndex)<br>Returns the Class for all Object instances in the specified column. |
| String | getColumnName(int columnIndex)<br>Returns the name of the specified column. |
| TableModelListener[] | getTableModelListeners()<br>Returns an array containing the listeners that have been added to the table model. |
| boolean | isCellEditable(int rowIndex, int columnIndex)<br>Returns true if the specified cell is editable, and false if it is not. |
| void | removeTableModelListener(TableModelListener listener)<br>Removes a listener from the table model so that it will no longer receive notification of changes |
| void | setValueAt(Object value, int rowIndex, int columnIndex)<br>Sets the value of the given cell. |

**No implementation of**

**getColumnCount()**

public interface **TableModel**

A TableModel is a two dimensional data structure that can store arbitrary Object instances, usually for the purp
with it.

The DefaultTableModel class provides one implementation of this interface.

## Method Summary

| | |
|---|---|
| void | addTableModelListener(TableModelListener listener)<br>Adds a listener to the model. |
| Class | getColumnClass(int columnIndex)<br>Returns the Class for all Object instances in the specified column. |
| int | getColumnCount()<br>Returns the number of columns in the model. |
| String | getColumnName(int columnIndex)<br>Returns the name of a column in the model. |
| int | getRowCount()<br>Returns the number of rows in the model. |
| Object | getValueAt(int rowIndex, int columnIndex)<br>Returns the value (Object) at a particular cell in the table. |
| boolean | isCellEditable(int rowIndex, int columnIndex)<br>Returns true if the cell is editable, and false otherwise. |
| void | removeTableModelListener(TableModelListener listener)<br>Removes a listener from the model. |
| void | setValueAt(Object aValue, int rowIndex, int columnIndex)<br>Sets the value at a particular cell in the table. |

```
46: /**
47:  * A two dimensional data structure used to store <code>Object</code>
48:  * instances, usually for display in a <code>JTable</code> component.
49:  *
50:  * @author    Andrew Selkirk
51:  */
52: public class DefaultTableModel extends AbstractTableModel
53:   implements Serializable
```

```
  /**
   * Returns the number of columns in the model.
   *
   * @return The column count.
   */
  public int getColumnCount()
  {
    return columnIdentifiers == null ? 0 : columnIdentifiers.size();
  }
```

# Singleton

**java.lang**

# Class Runtime

java.lang.Object
  └ **java.lang.Runtime**

---

public class **Runtime**
extends Object

Every Java application has a single instance of class `Runtime` that allows the application to interface with the environment in which the application is running. The current runtime can be obtained from the `getRuntime` method.

An application cannot create its own instance of this class.

**Since:**
JDK1.0
**See Also:**
getRuntime()

---

## Method Summary

| | |
|---|---|
| void | **addShutdownHook**(Thread hook)<br>Registers a new virtual-machine shutdown hook. |
| int | **availableProcessors**()<br>Returns the number of processors available to the Java virtual machine. |
| Process | **exec**(String command)<br>Executes the specified string command in a separate process. |
| Process | **exec**(String[] cmdarray)<br>Executes the specified command and arguments in a separate process. |
| Process | **exec**(String[] cmdarray, String[] envp)<br>Executes the specified command and arguments in a separate process with the specified environment. |
| Process | **exec**(String[] cmdarray, String[] envp, File dir)<br>Executes the specified command and arguments in a separate process with the specified environment and working directory. |
| Process | **exec**(String command, String[] envp)<br>Executes the specified string command in a separate process with the specified environment. |
| Process | **exec**(String command, String[] envp, File dir)<br>Executes the specified string command in a separate process with the specified environment and working directory. |
| void | **exit**(int status)<br>Terminates the currently running Java virtual machine by initiating its shutdown sequence. |

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

### getRuntime

```
public static Runtime getRuntime()
```

Returns the runtime object associated with the current Java application. Most of the methods of class `Runtime` are instance methods and must be invoked with respect to the current runtime object.

**Returns:**

the `Runtime` object associated with the current Java application.

java.awt

# Class Desktop

java.lang.Object
  └ **java.awt.Desktop**

---

public class **Desktop**
extends Object

The `Desktop` class allows a Java application to launch associated applications registered on the native desktop to handle a `URI` or a file.

Supported operations include:

- launching the user-default browser to show a specified URI;
- launching the user-default mail client with an optional `mailto` URI;
- launching a registered application to open, edit or print a specified file.

This class provides methods corresponding to these operations. The methods look for the associated application registered on the current platform, and launch it to handle a URI or file. If there is no associated application or the associated application fails to be launched, an exception is thrown.

An application is registered to a URI or file type; for example, the `"sxi"` file extension is typically registered to StarOffice. The mechanism of registereing, accessing, and launching the associated application is platform-dependent.

Each operation is an action type represented by the `Desktop.Action` class.

Note: when some action is invoked and the associated application is executed, it will be executed on the same system as the one on which the Java application was launched.

## getDesktop

public static Desktop **getDesktop**()

Returns the `Desktop` instance of the current browser context. On some platforms the Desktop API may not be supported; use the `isDesktopSupported()` method to determine if the current desktop is supported.

**Returns:**
the Desktop instance of the current browser context
**Throws:**
`HeadlessException` - if `GraphicsEnvironment.isHeadless()` returns `true`
`UnsupportedOperationException` - if this class is not supported on the current platform
**See Also:**
`isDesktopSupported()`, `GraphicsEnvironment.isHeadless()`

# « Singleton »

- Applicability
  - There must be exactly one instance of a class
  - When it must be accessible to clients from a well-known place
  - When the sole instance should be extensible by subclassing, with unmodified clients using the subclass
- Consequences
  - Controlled access to sole instance
  - Reduced name space (vs. global variables)
  - Can be refined in subclass or changed to allow multiple instances

# Implementation of Singleton

- Constructor is protected
- Instance variable is private
- Public operation returns singleton
  - May lazily create singleton
- Subclassing
  - Instance() method can look up subclass to create in environment

```
public final class LazySingleton {
    private static volatile LazySingleton instance = null;

    // private constructor
    private LazySingleton() {
    }

    public static LazySingleton getInstance() {
        if (instance == null) {
            synchronized (LazySingleton.class) {
                instance = new LazySingleton();
            }
        }
        return instance;
    }
}
```

# Implementation of Singleton

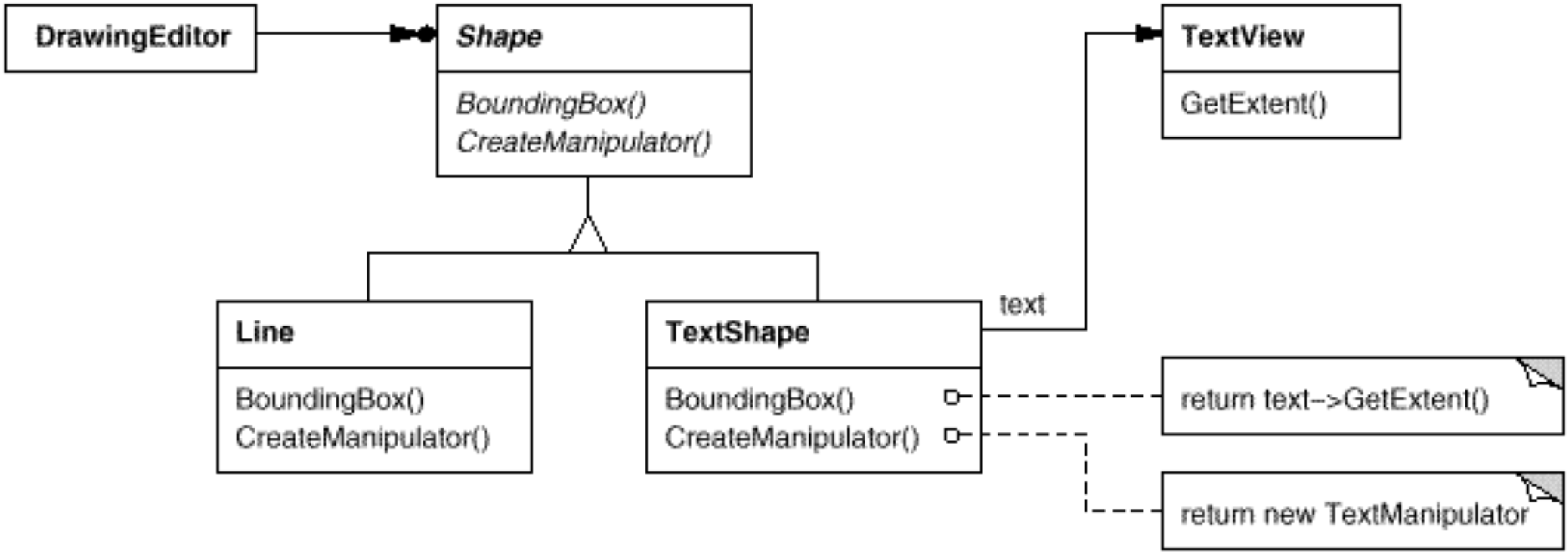
- Constructor is protected, Instance variable is private

```
public class StaticBlockSingleton {
    private static final StaticBlockSingleton INSTANCE;

    static {
        try {
            INSTANCE = new StaticBlockSingleton();
        } catch (Exception e) {
            throw new RuntimeException("Uffff, i was not expecting this!", e);
        }
    }

    public static StaticBlockSingleton getInstance() {
        return INSTANCE;
    }

    private StaticBlockSingleton() {
        // ...
    }
}
```

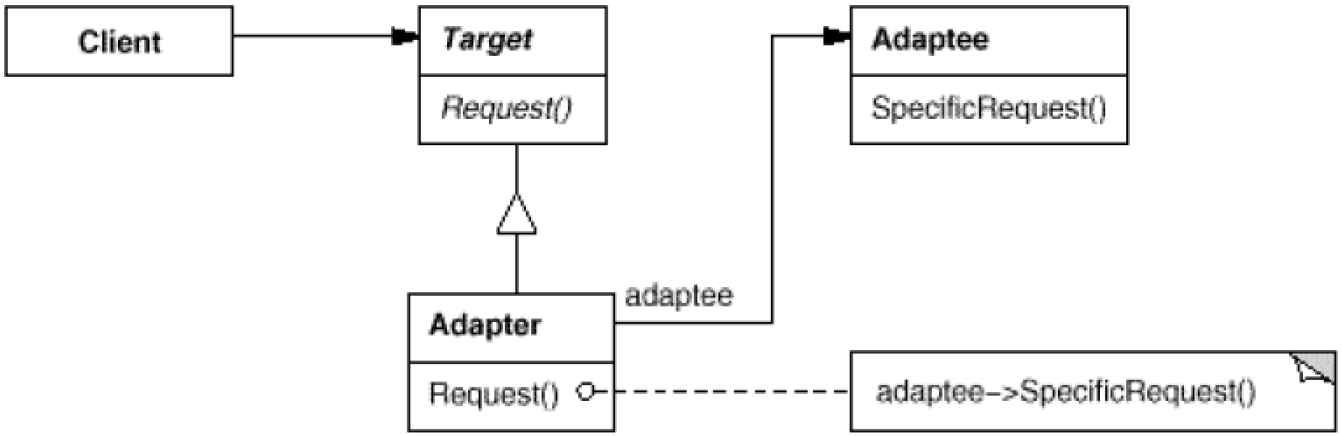
```
class FileSystem
{
public:
  static FileSystem& instance()
  {
    // Lazy initialize.
    if (instance_ == NULL) instance_ = new FileSystem();
    return *instance_;
  }

private:
  FileSystem() {}

  static FileSystem* instance_;
};
```

# Adapter

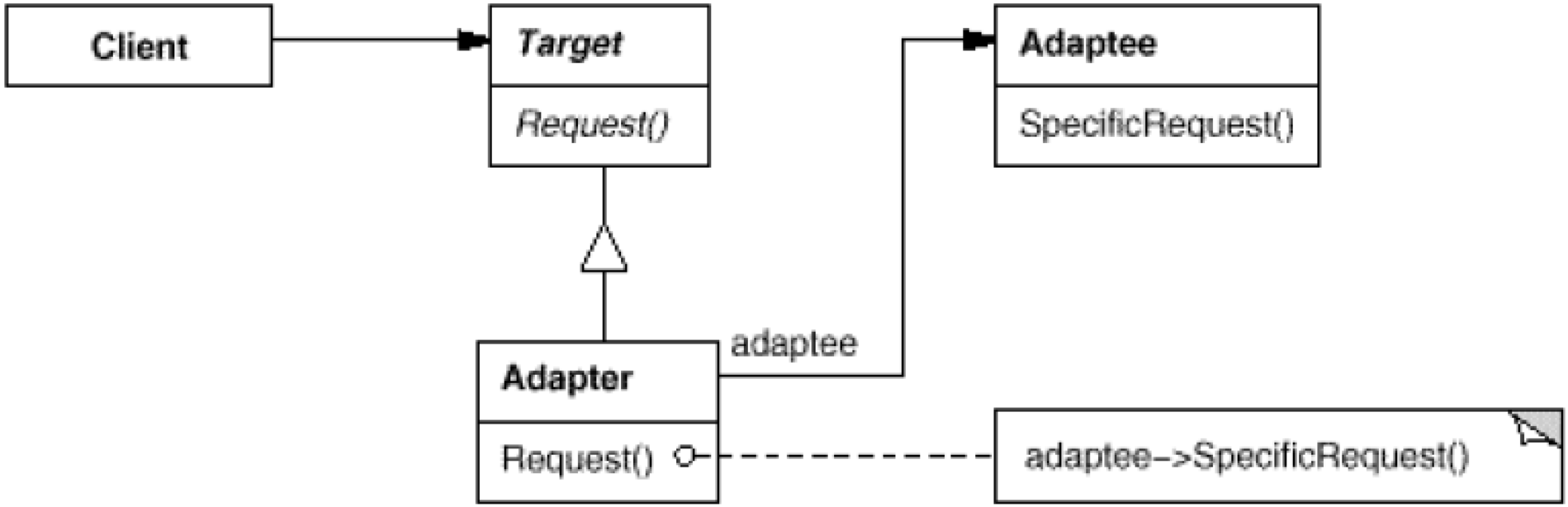
# « Adapter » aka Wrapper (Mariage de convenance)



- Applicability
  - You want to use an existing class, and its interface does not match the one you need
  - You want to create a reusable class that cooperates with unrelated classes that don't necessarily have compatible interfaces
  - You need to use several subclasses, but it's impractical to adapt their interface by subclassing each one
- Consequences
  - Exposes the functionality of an object in another form
  - Unifies the interfaces of multiple incompatible adaptee objects
  - Lets a single adapter work with multiple adaptees in a hierarchy

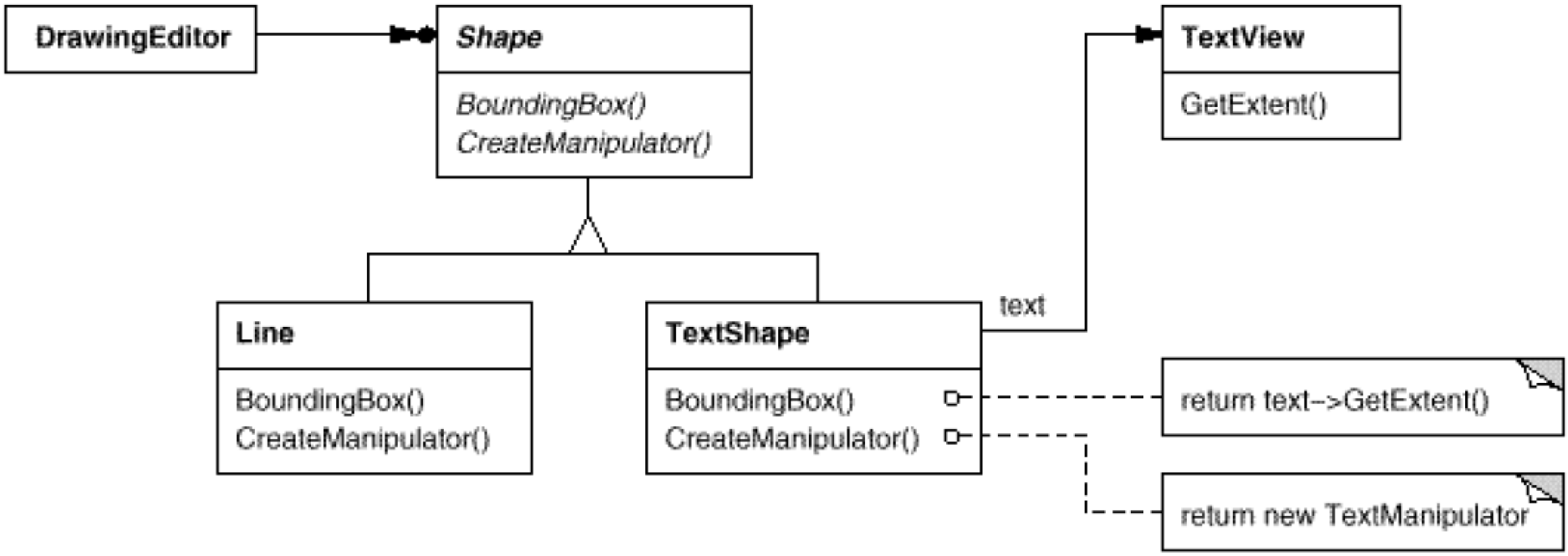


- **Structure (version par délégation)**




- **Version par héritage ⇒ nécessite héritage multiple ou héritage d'implémentation**

- **Participants**
  - Target (Shape) définit l'interface spécifique à l'application que le client utilise
  - Client (DrawingEditor) collabore avec les objets qui sont conformes à l'interface de Target
  - Adaptee (TextView) est l'interface existante qui a besoin d'adaptation
  - Adapter (TextShape) adapte effectivement l'interface de Adaptee à l'interface de Target



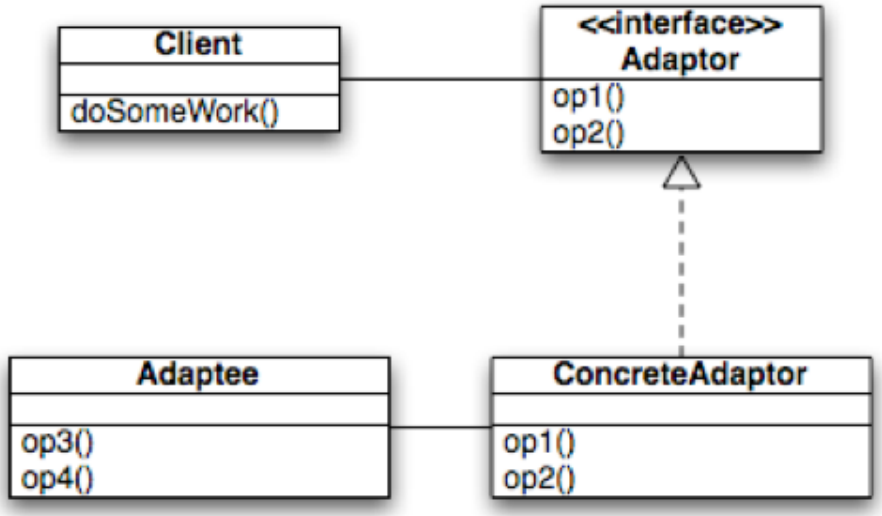
# Patron « Adapter »

- Objectif :
  - Permet le réemploi d'un type qui n'est pas conforme à une interface attendue
- Exemple
  - votre code emploie une interface Stack « idéale » (push, pop, top, size)
  - la classe Java disponible n'a pas exactement cette interface

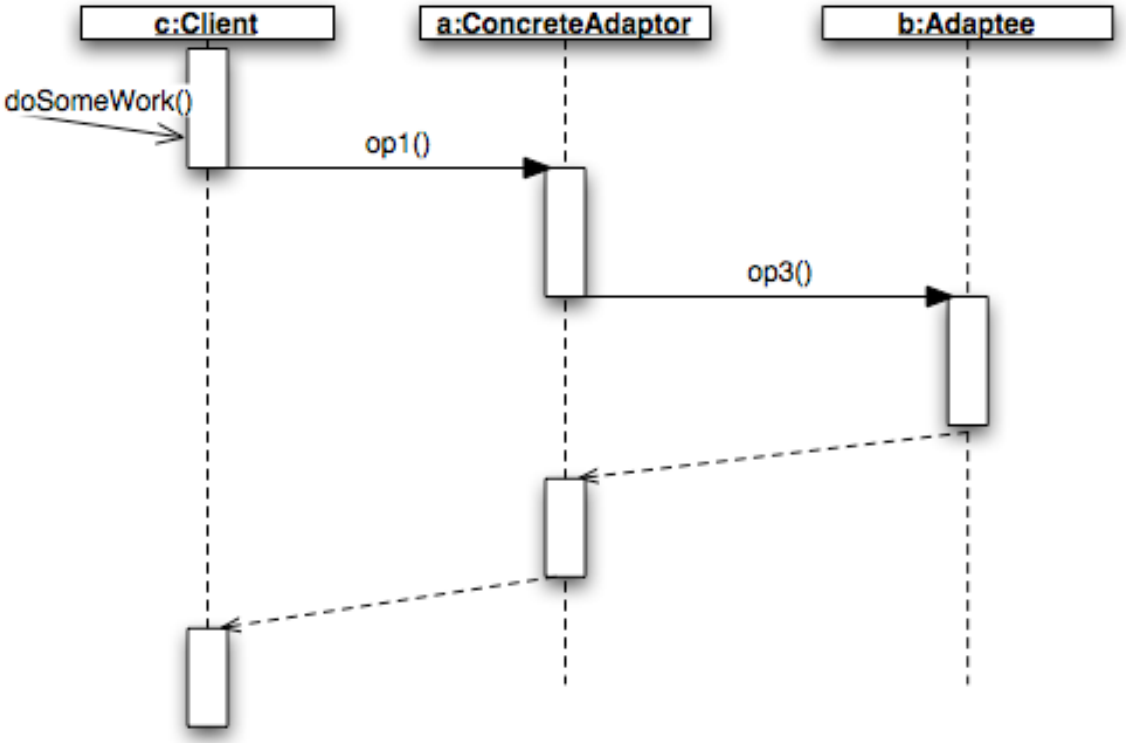
# Patron « Adapter » - rôles

- Client
  - emploie des opérations de l'interface Adaptor
- Adaptor
  - définit les opérations attendues par le Client
- ConcreteAdaptor
  - réalise les opérations par délégation vers Adaptee
- Adaptee
  - contient la mise en œuvre à réutiliser

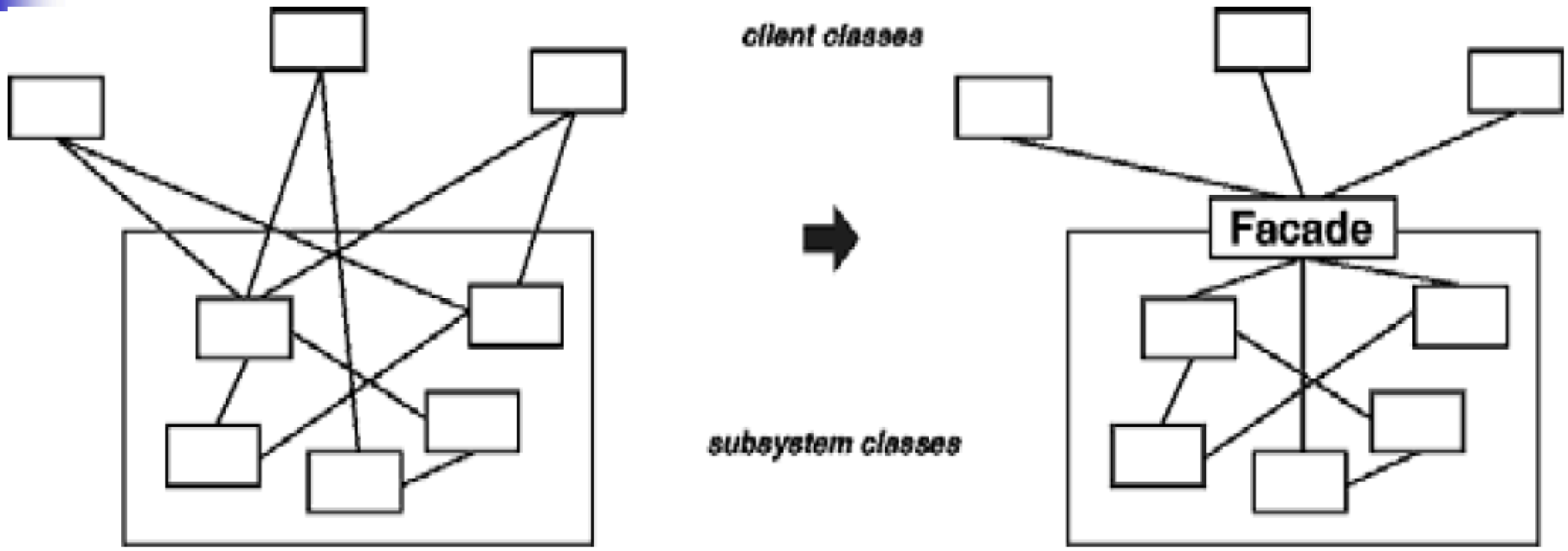
# Patron « Adapter » - structure

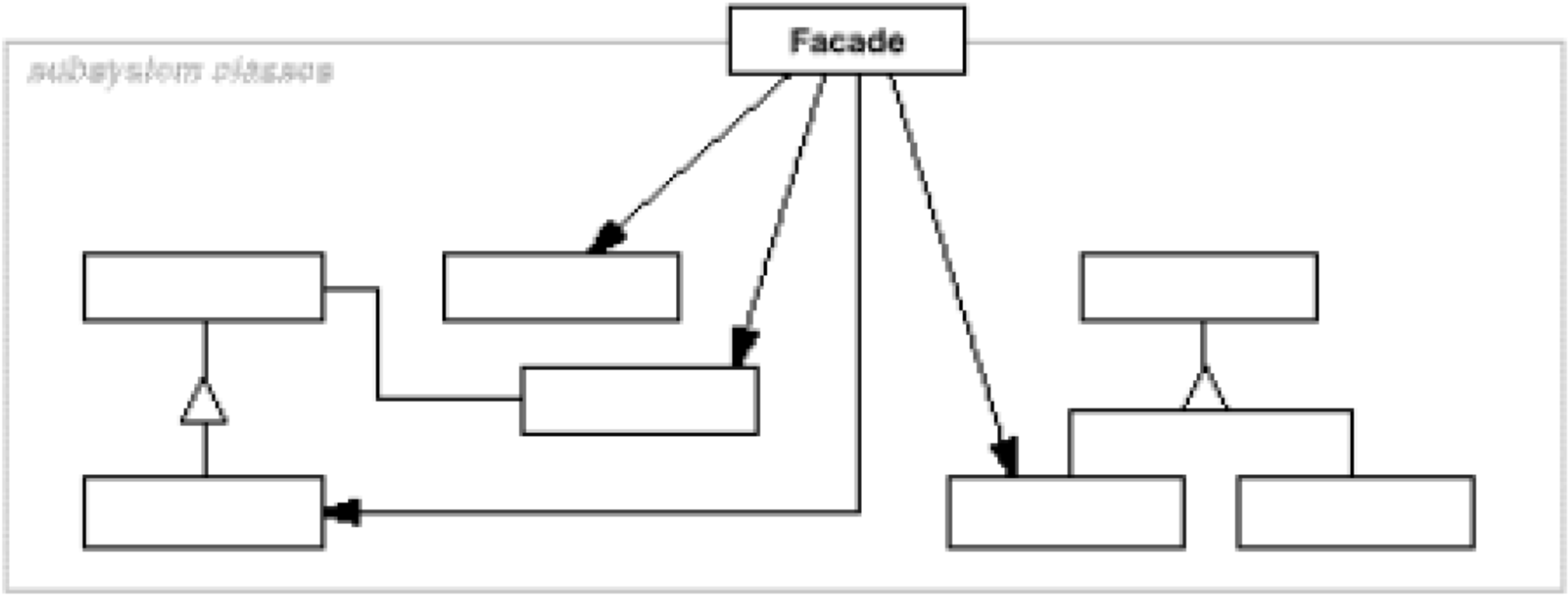


# Patron « Adapter » - diagramme de séquence

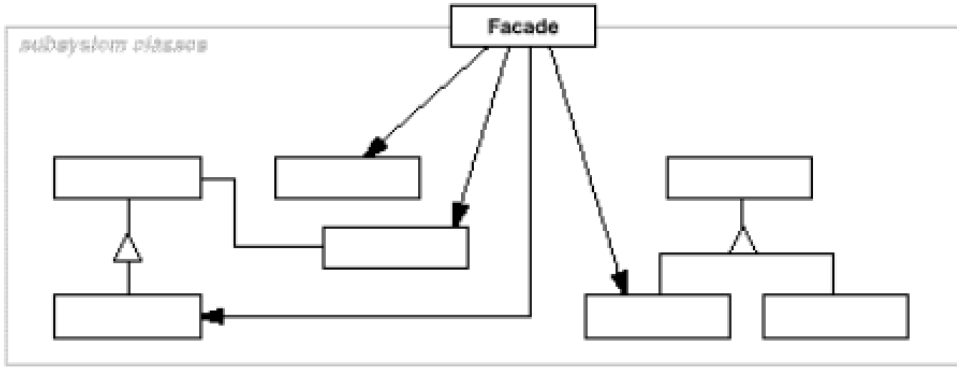


# Facade


client classes
Facade
subsystem classes


Facade
subsystem classes

# javax.faces.context.ExternalContext




**javax.faces.context**

## Class ExternalContext

```
java.lang.Object
  └ javax.faces.context.ExternalContext
```

**Direct Known Subclasses:**
ExternalContextWrapper

---

```
public abstract class ExternalContext
extends java.lang.Object
```

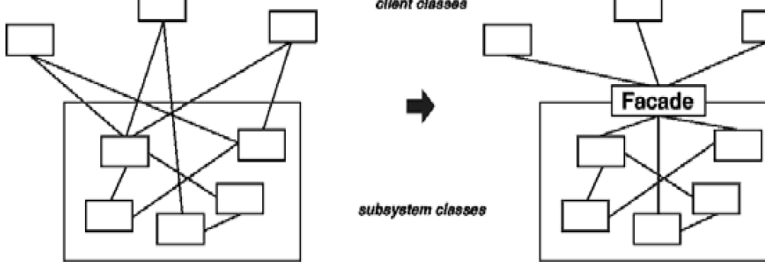
This class allows the Faces API to be unaware of the nature of its containing application environment. In particular, this class allows JavaServer Faces based appications to run in either a Servlet or a Portlet environment.

The documentation for this class only specifies the behavior for the *Servlet* implementation of `ExternalContext`. The *Portlet* implementation of `ExternalContext` is specified under the revision of the Portlet Bridge Specification for JavaServer Faces JSR that corresponds to this version of the JSF specification. See the Preface of the "prose document", linked from the javadocs, for a reference.

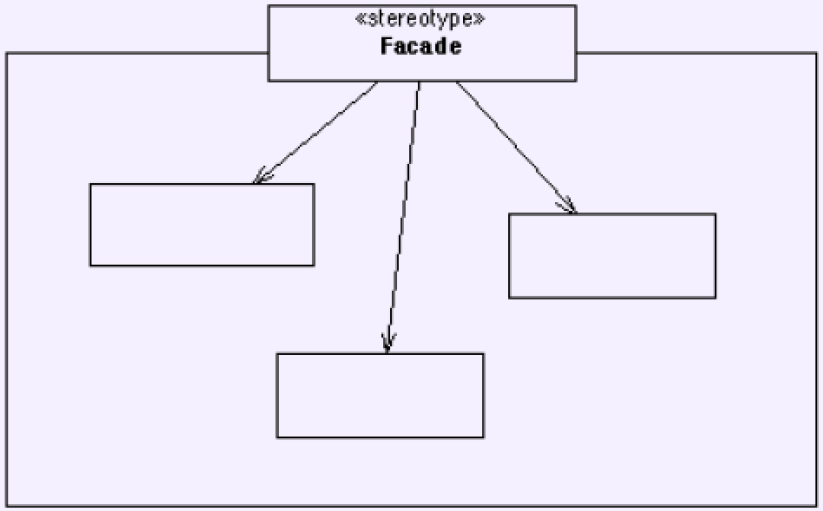
If a reference to an `ExternalContext` is obtained during application startup or shutdown time, any method documented as "valid to call this method during application startup or shutdown" must be supported during application startup or shutdown time. The result of calling a method during applicatic startup or shutdown time that does not have this designation is undefined.

internally uses ServletContext, HttpSession, HttpServletRequest, HttpServletResponse, etc.

# « Facade »




- Applicability
  - You want to provide a simple interface to a complex subsystem
  - You want to decouple clients from the implementation of a subsystem
  - You want to layer your subsystems




- Consequences
  - It shields clients from the complexity of the subsystem, making it easier to use
  - Decouples the subsystem and its clients, making each easier to change
  - Clients that need to can still access subsystem classes

# Facade vs Adapter

- Motivation
  - Modular decomposition: separate the client from subsystem/ Adaptee
  - Facade: simplify the interface (a new interface to the library)
  - Adapter: match an existing interface (adapt to the legacy)
- Adapter: interface is given (constraint)
  - Not typically true in Facade
- Adapter (polymorphic)
  - Dispatch dynamically to multiple implementations
  - Typically choose the implementation statically

# Abstract Factory

# Patron « Abstract Factory »

- L'objectif est de
  - permettre de créer des familles de produits
  - masquer les mécanismes de choix des classes de mise en œuvre de ces produits

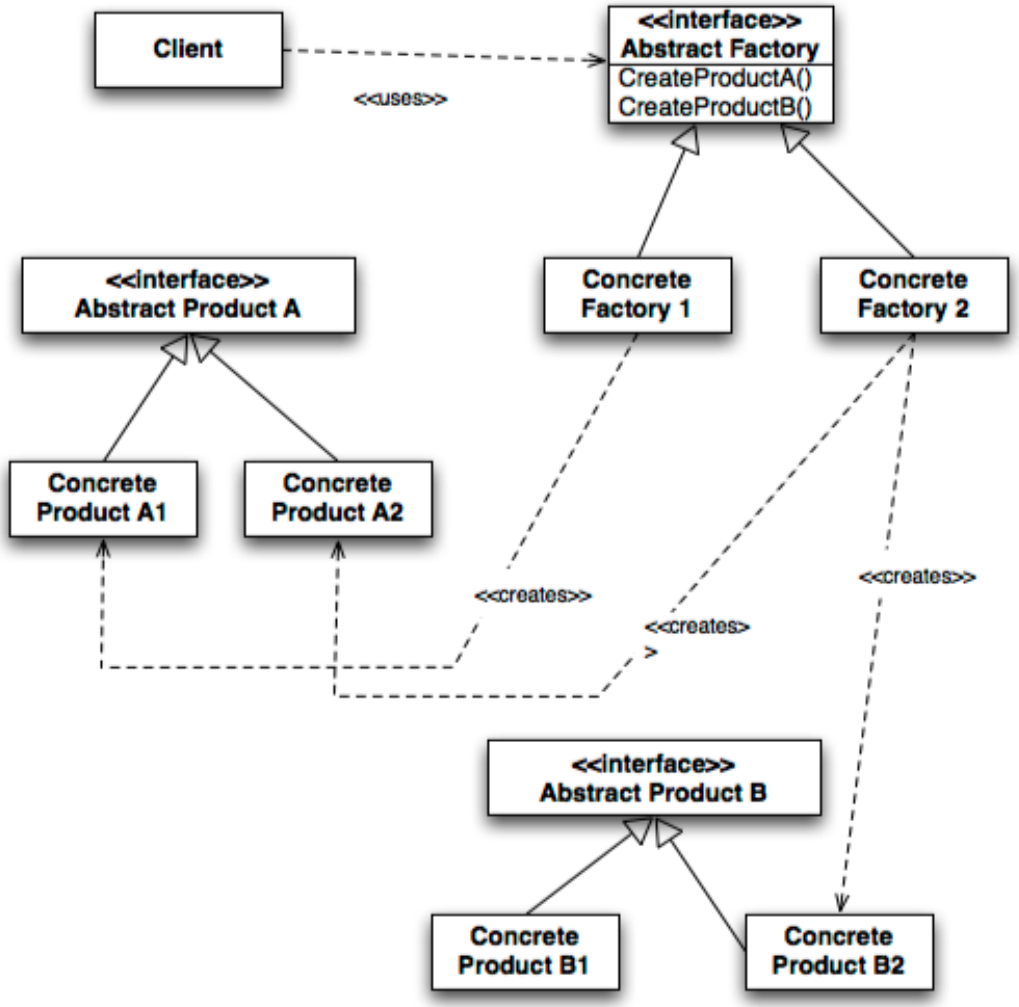
# Patron « Abstract Factory » - rôles

- Client
  - Détient une référence sur une Abstract factory
  - Crée des produits par appel des opérations de cette référence
  - Ne connaît pas la classe concrète des produits
- Abstract Product
  - Masquer la classe concrète
  - Offrir un ensemble d'opérations applicables à tous les variantes d'un même produit
- Abstract Factory
  - Comporte une opération de création (pour chaque produit, une opération de création retourne un objet produit)
  - La classe concrète des produits est masquée

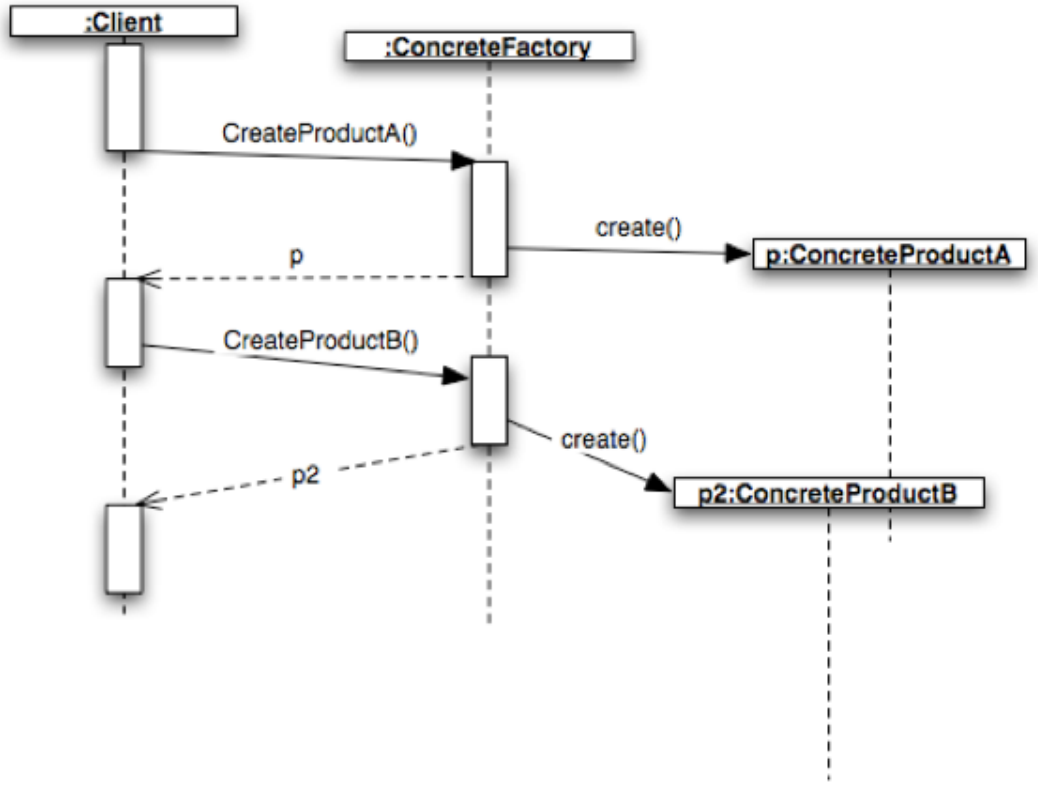
# Patron « Abstract Factory » - rôles

- Concrete Product
  - Contient la mise en œuvre spécifique des opérations
  - Non accessible au client
  - Peut être amené à jouer un rôle d'adaptateur
- Concrete Factory
  - Chargée de mettre en œuvre la création des produits concrets
  - Une fabrique concrète pour une plate-forme/variante/version donnée ne fait que des produits concrets de la même plate-forme/variante/version

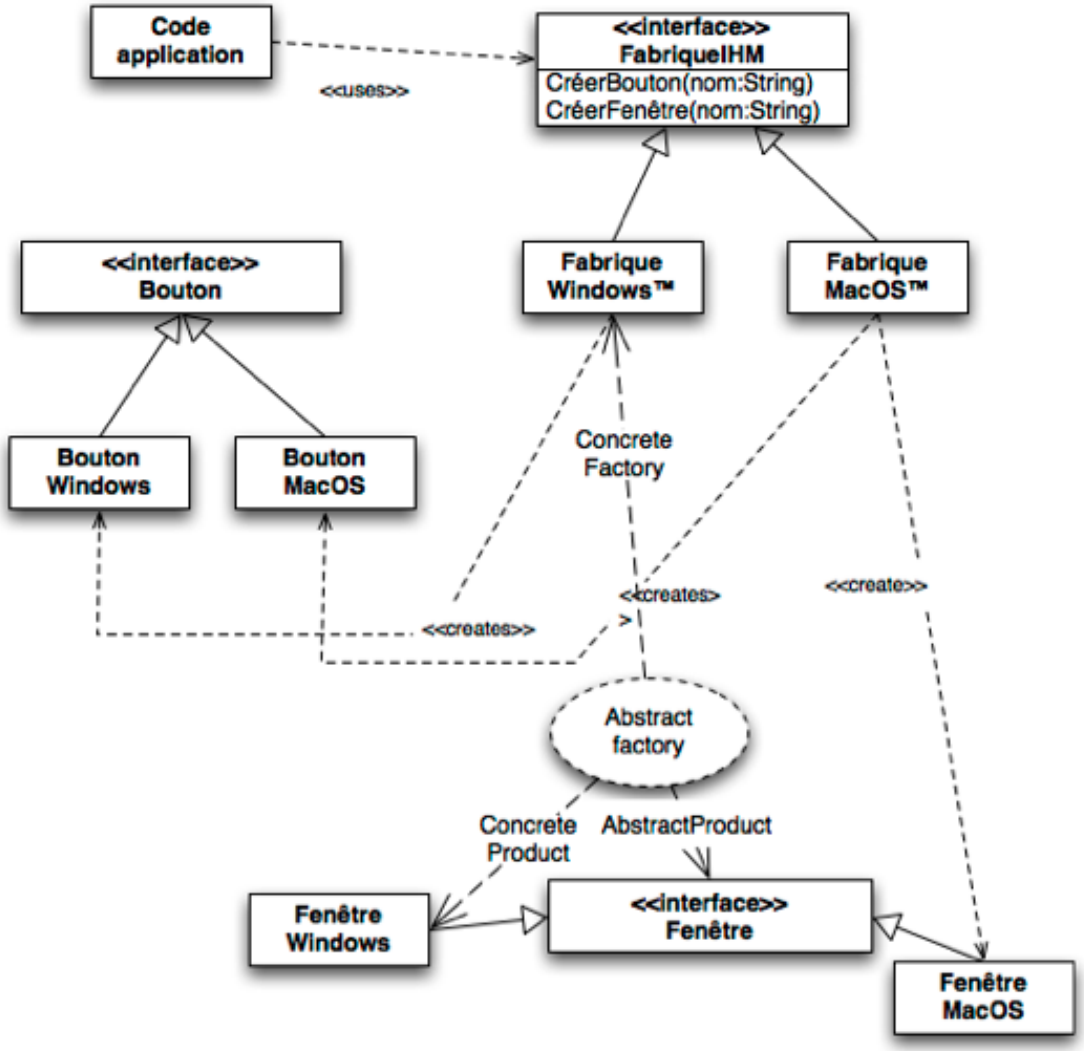
# Patron « Abstract Factory » - structure



# Patron « Abstract Factory » - diagramme de séquence



# Patron « Abstract Factory » - example



# Visitor

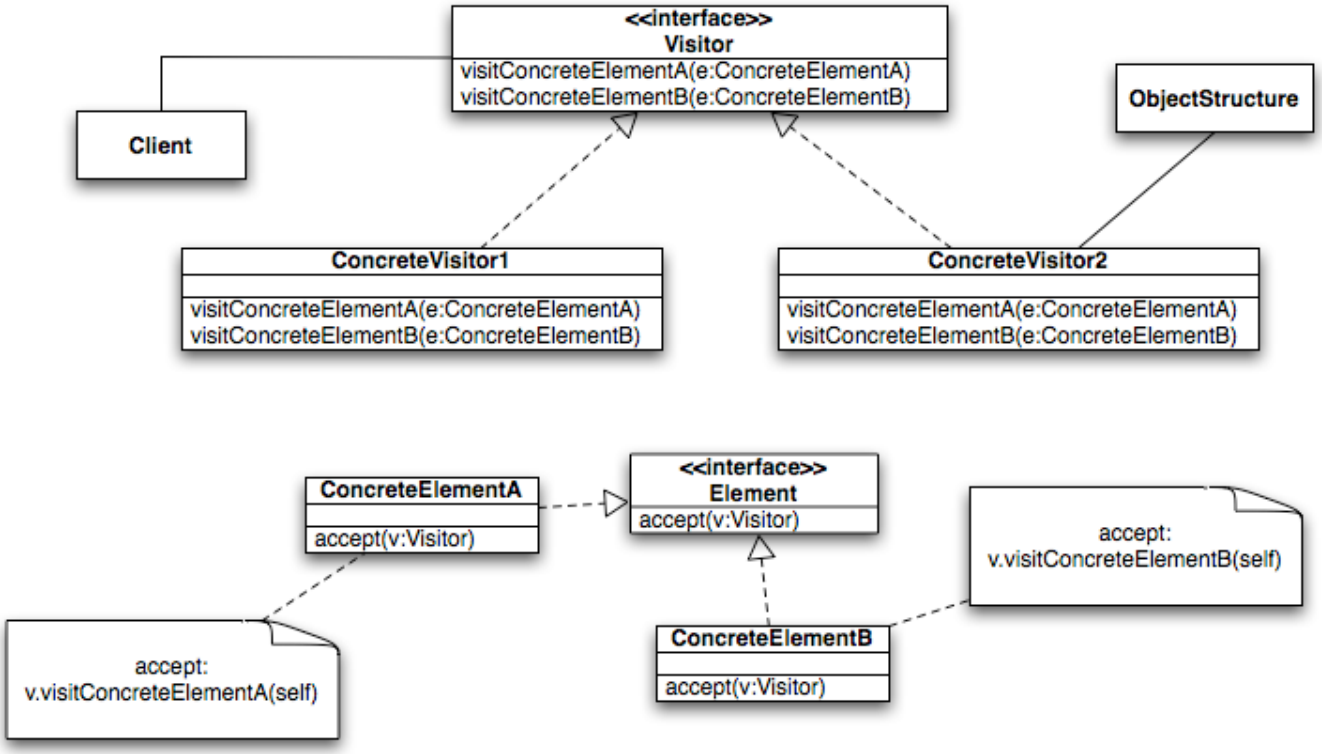
# Patron « Visitor »

- Objectif :
  - faciliter l'organisation des méthodes de traitement d'une structure de graphe/d'arbre
  - séparer le choix des techniques de traitement de la description des types
  - permet d'étendre ou de modifier les traitements en ne changeant qu'un fichier source

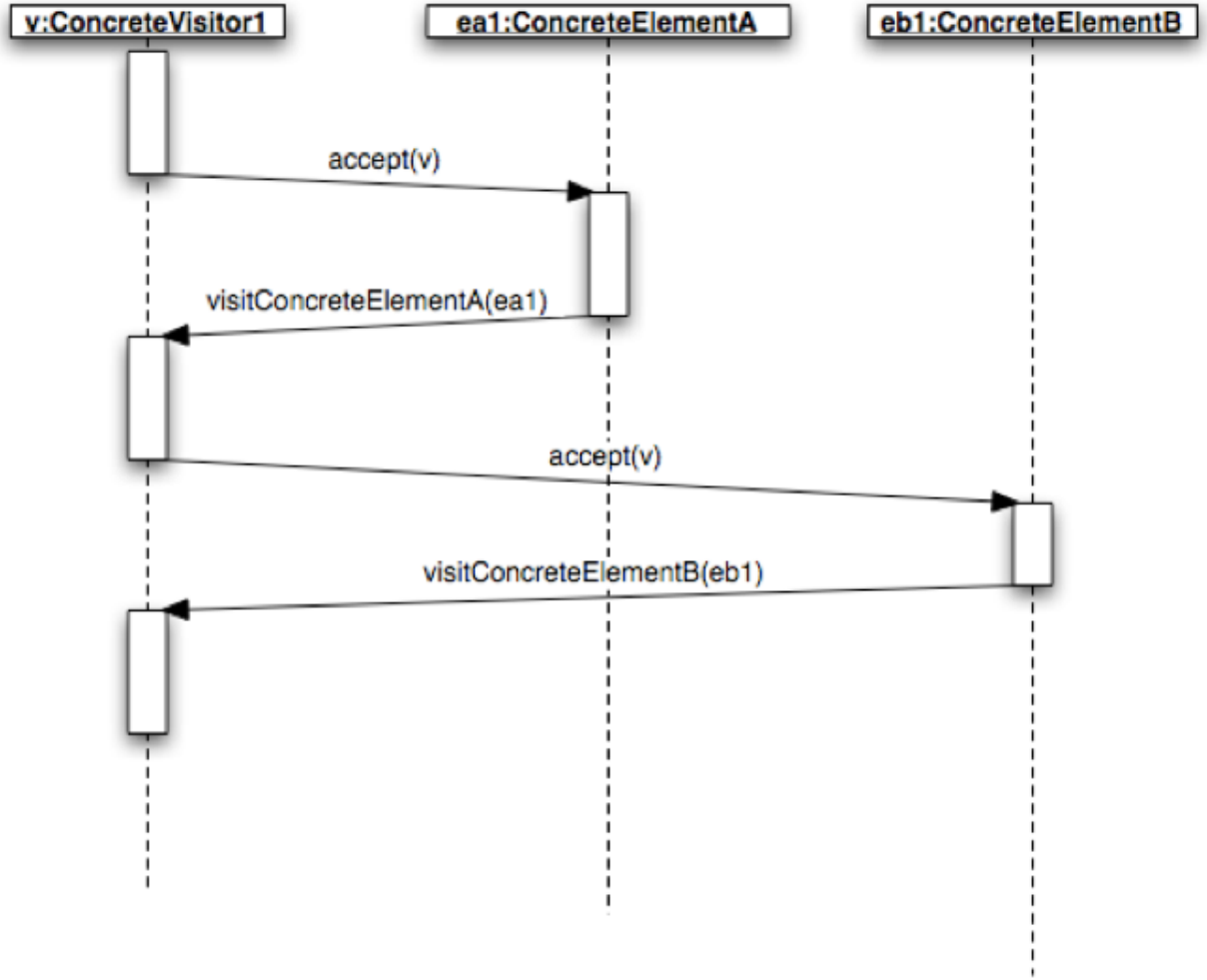
# Patron « Visitor » - rôles

- Visitor
  - Définir une opération dite de traitement pour chaque type d'élément (*visitFoo()*)
  - Cette opération sera appelée par chaque élément pour déclencher le traitement qui le concerne
- Element
  - Déclare une opération qui sera appelée par le Visitor (*accept()*)
- ConcreteElement
  - Mise en œuvre de *Element::accept()*
  - Le but est de déclencher le traitement correct

# Patron « Visitor » - structure



# Patron « Visitor » - diagramme de séquence



# Patron « Visitor »

- Remarque :
  - Le contrôle du parcours est sous la responsabilité du visiteur, pas des éléments
- Quand utiliser le patron « Visitor »
  - Pour parcourir des structures d'éléments (graphes, arbres)
  - Quand l'ensemble des types change peu souvent
    - si nouveau type : il faut modifier tous les visiteurs
  - Quand les traitements changent souvent
    - une seule classe à ajouter ou à modifier

# Memento

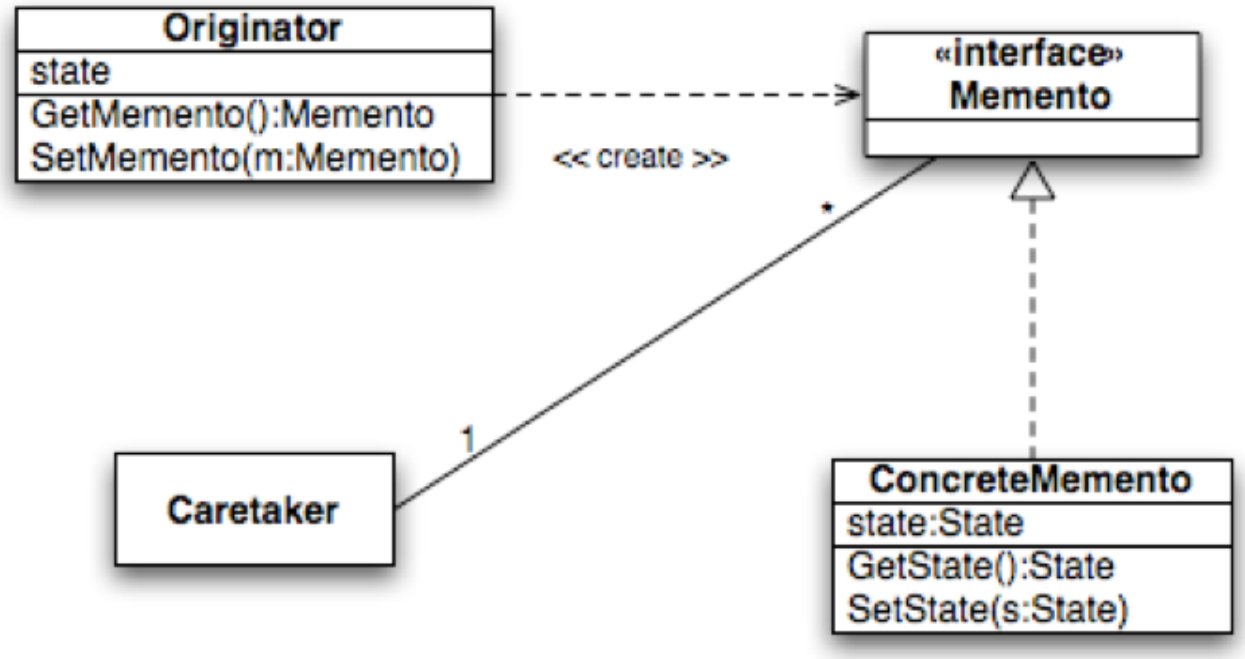
# Patron « Memento »

- L'objectif est de capturer l'état d'un objet pour le stocker et le restaurer plus tard, sans briser l'encapsulation
- Principe : une interface Memento sert de type opaque

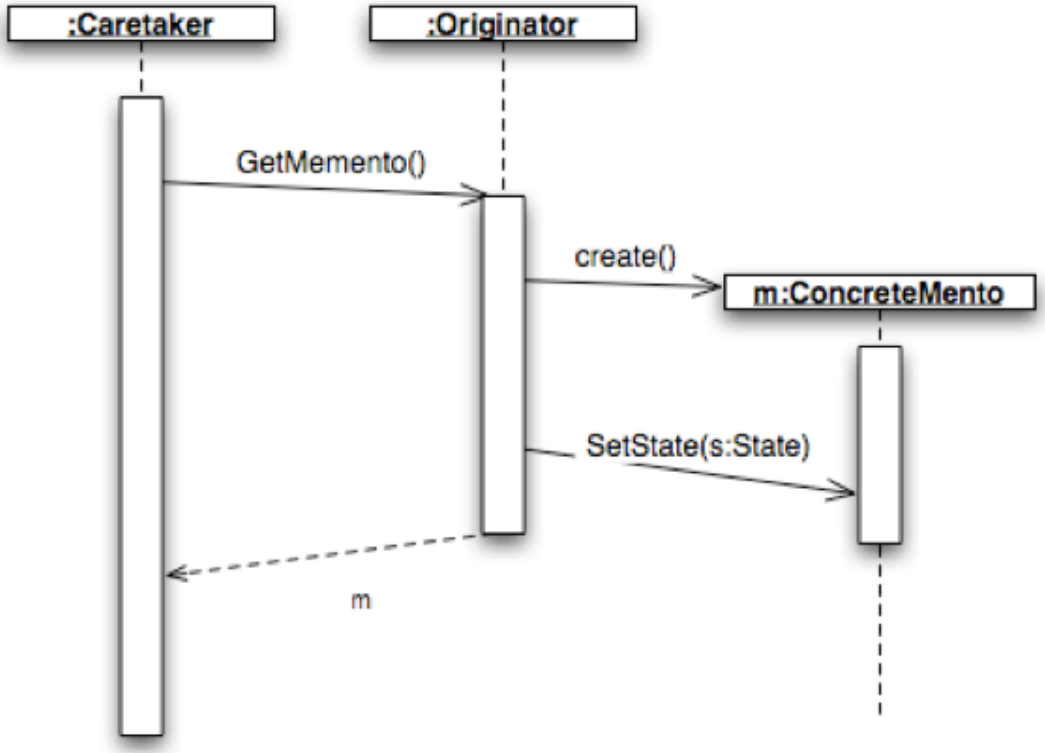
# Patron « Memento » - rôles

- Originator
  - possède un état à sauver/restaurer
  - est capable de créer des mémentos concrets
- Memento
  - interface permettant de transmettre des états sauvegardés de manière opaque
- ConcreteMemento
  - mise en œuvre de stockage d'un état
- Caretaker
  - capable de stocker des mémentos et de les récupérer

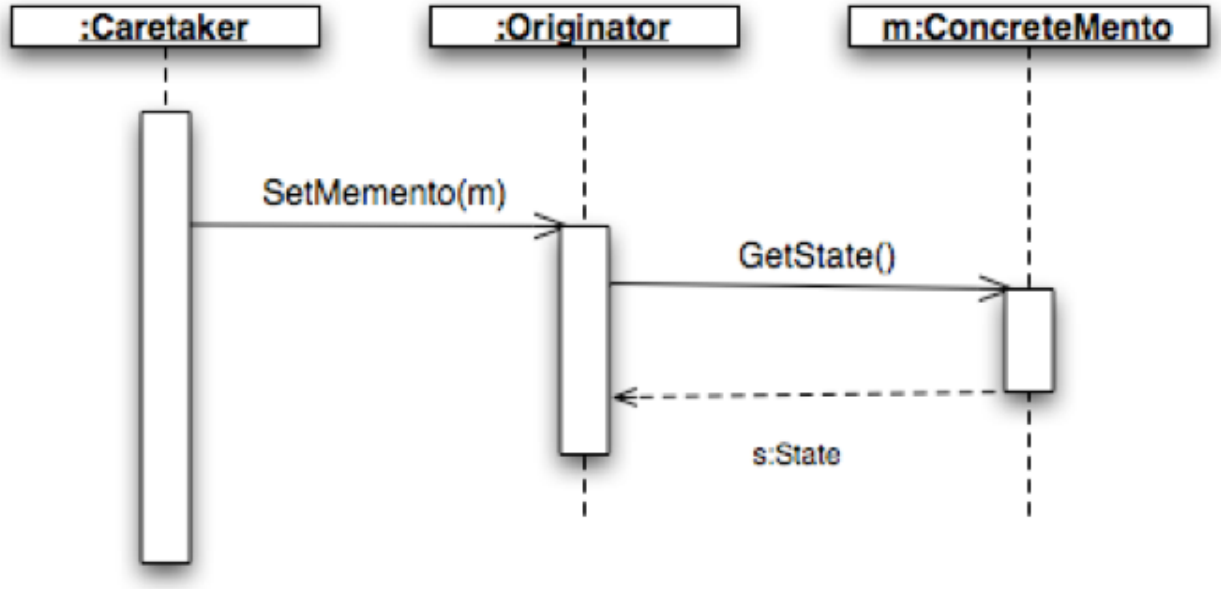
# Patron « Memento » - structure



# Patron « Memento » - sauvegarde



# Patron « Memento » - restauration

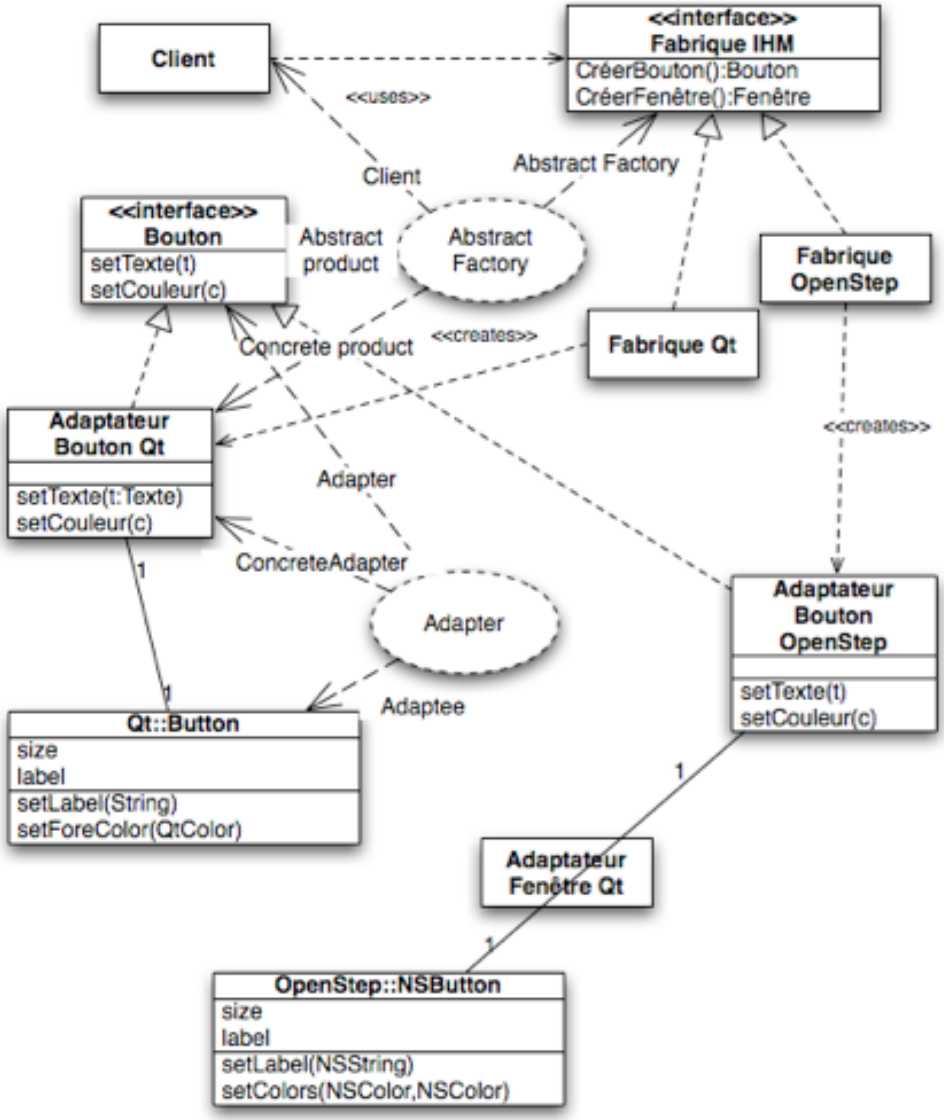


# Adapter
# +
# Abstract Factory

# Associer « Adapter » et « Abstract Factory »

- Les différentes fabriques concrètes existantes
  - créent des produits ayant des interfaces différentes
  - doivent être réunies par un concept de produit abstrait
- Solution
  - on interpose un PC Adapter entre Abstract product et Concrete product

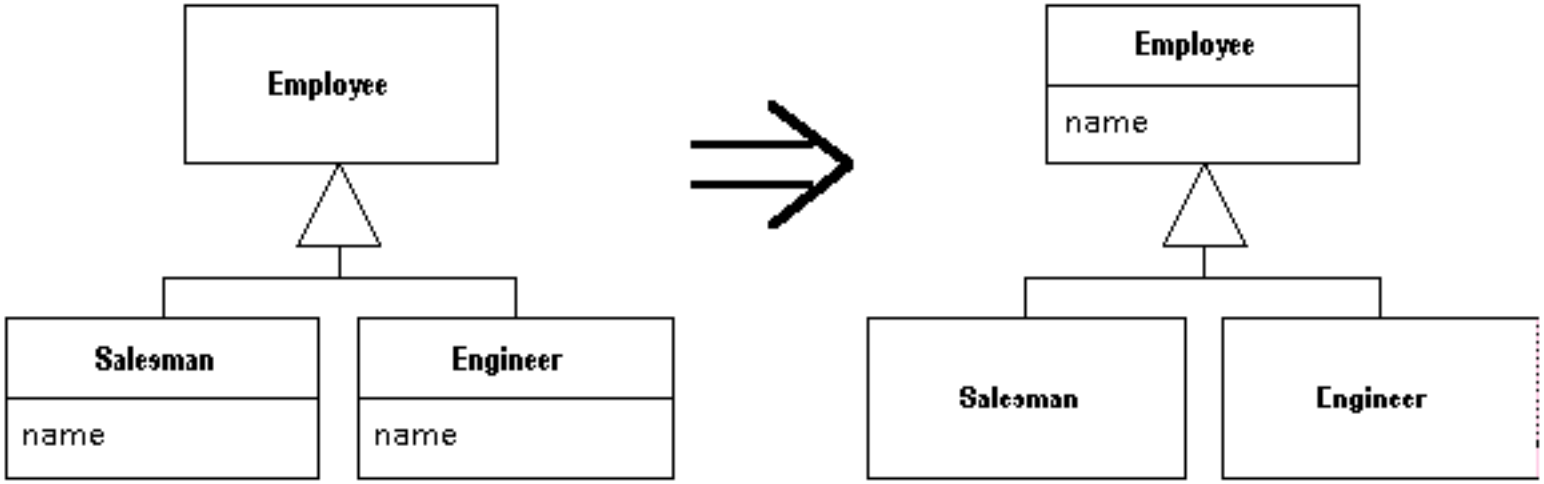
# Associer « Adapter » et « Abstract Factory »



# Refactorings

*Two subclasses have the same field.*

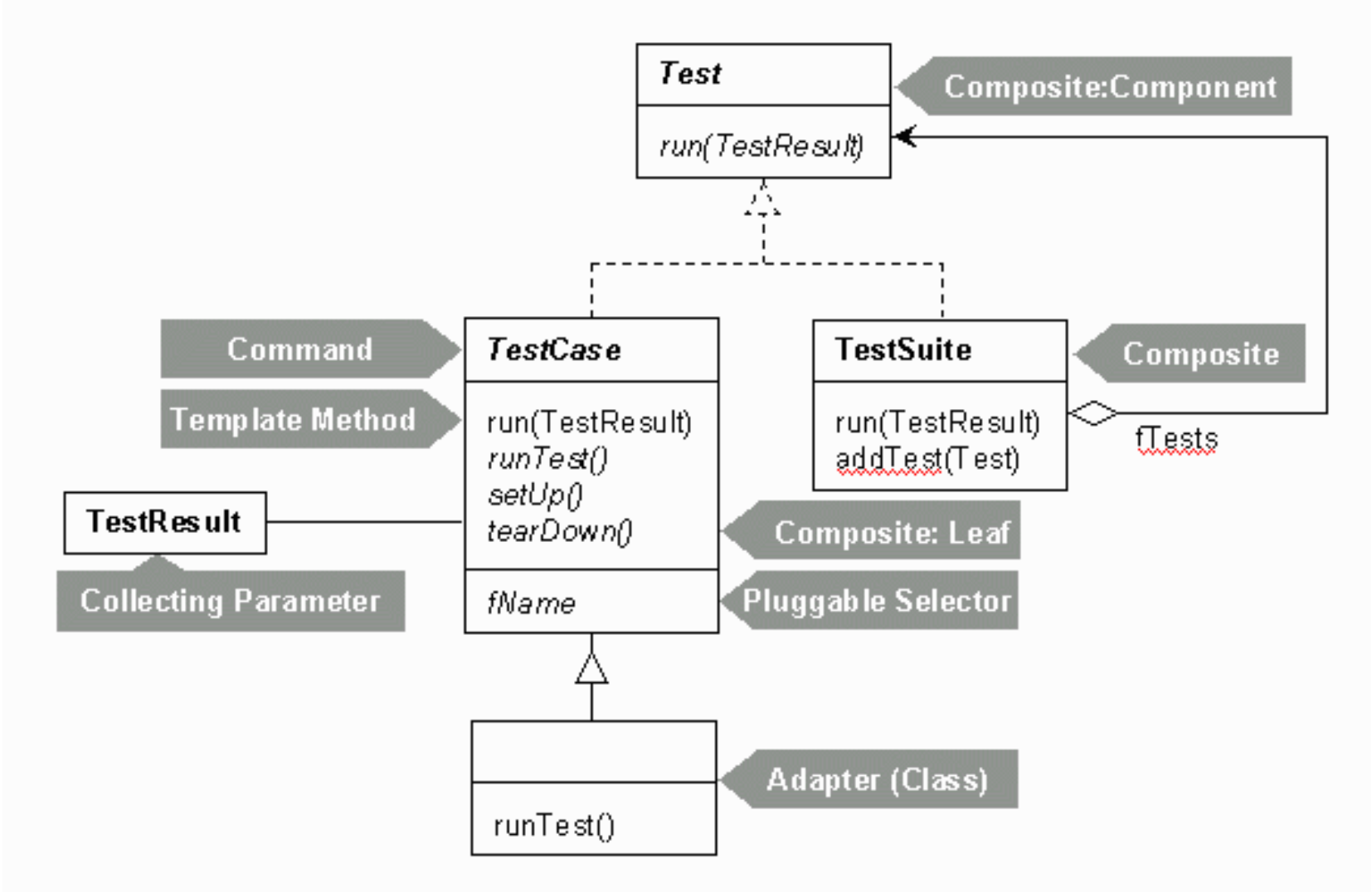
**Move the field to the superclass.**



# Exercise

- « Test » framework

# JUnit and… Design patterns




**Worth reading!**

**http://junit.sourceforge.net/doc/cookstour/cookstour.htm**

# Et dans le futur ? *(from E. Gamma)*

## a new categorization

- **Core**
  - Composite
  - Strategy
  - State
  - Command
  - Iterator
  - Proxy
  - Template Method
  - Facade
  - ***Null Object***

- **Creational**
  - Factory method
  - Prototype
  - Builder
  - ***Dependency Injection***
- **Peripheral**
  - Abstract Factory (peripheral)
  - Memento
  - Chain of responsibility
  - Bridge
  - Visitor
  - ***Type Object***
  - Decorator
  - Mediator
  - Singleton
  - ***Extension Objects***
- **Other** (Compound)
  - Interpreter
  - Flyweight

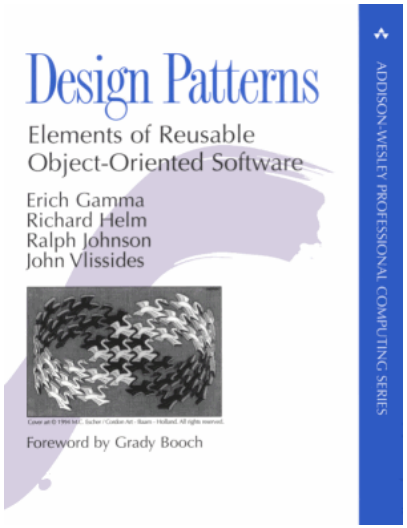
**the patterns the students should learn**

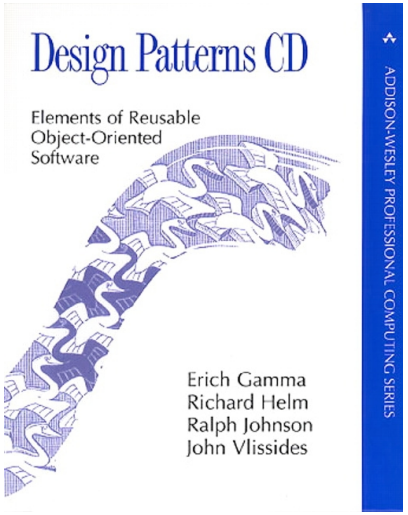
**lean on demand**

# Previously

- #1
  - Composite, State, Strategy, Command, Observer
- #2
  - Template Method, Singleton, Facade, Abstract Factory, Visitor, Memento, Adapter
- #3
  - Iterator

# References


Design Patterns
Elements of Reusable
Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES


Design Patterns CD
Elements of Reusable
Object-Oriented
Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES


A Brain-Friendly Guide
Head First
Design Patterns
O'REILLY
Eric Freeman & Elisabeth Freeman
with Kathy Sierra & Bert Bates

# References


http://refcardz.dzone.com/refcardz/design-patterns