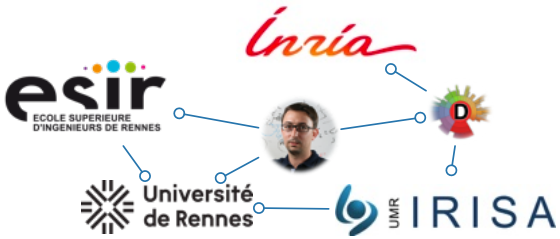
(Object-Oriented) Design Patterns

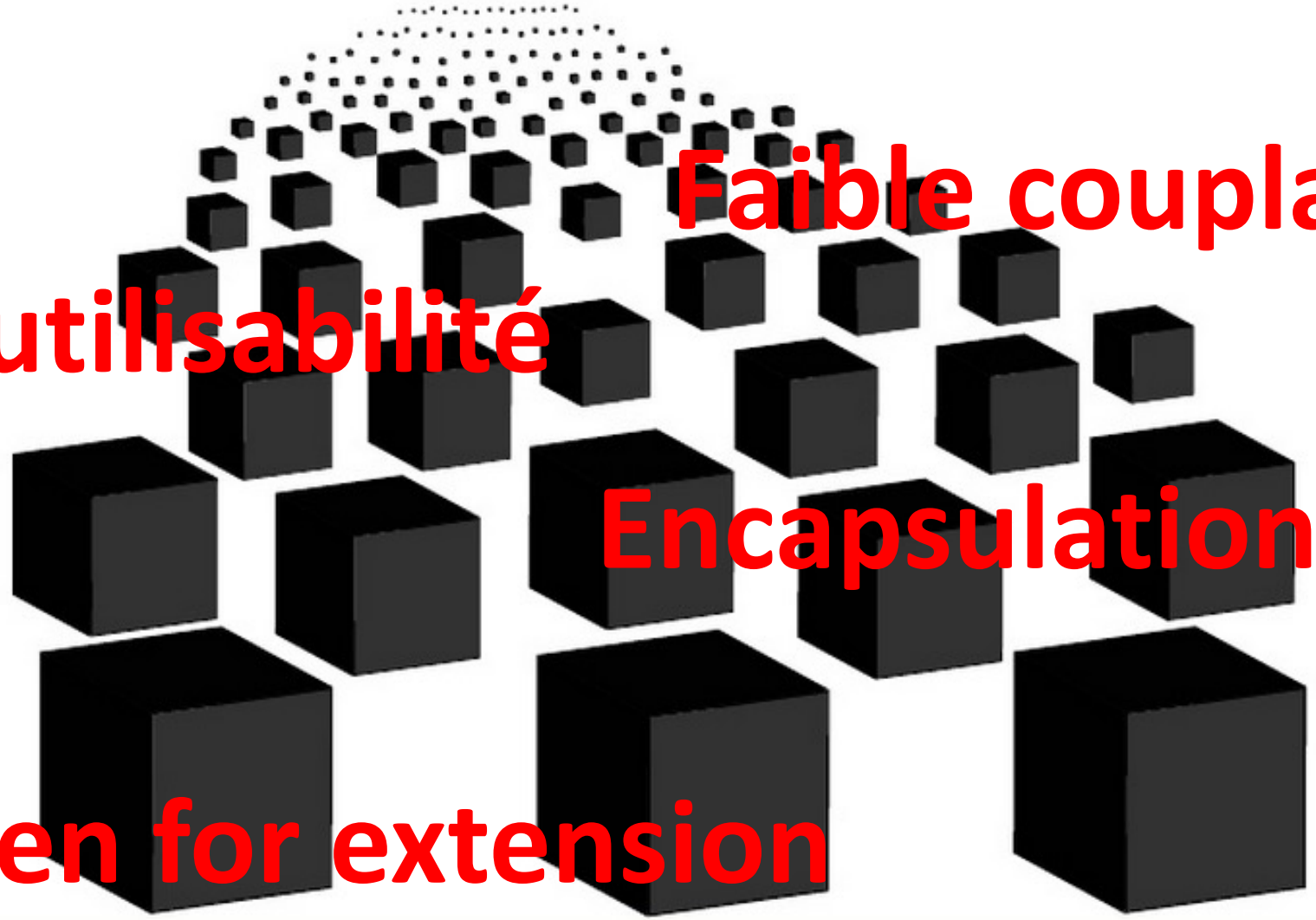
Material available at http://combemale.fr
Version Sep., 2023

BENOIT COMBEMALE
FULL PROFESSOR, UNIVERSITY OF RENNES, FRANCE

HTTP://COMBEMALE.FR
BENOIT.COMBEMALE@IRISA.FR
@BCOMBEMALE



Idéalement:
« modular black boxes »



Faible couplage

Réutilisabilité

Encapsulation

Open for extension

"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

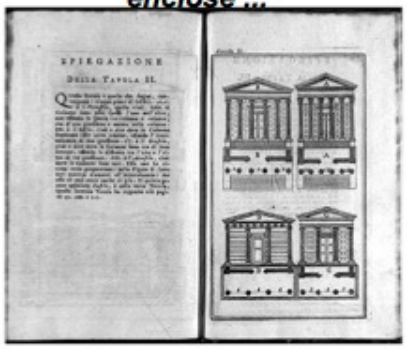
Clues from other disciplines *(from E. Gamma)*

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 ..."

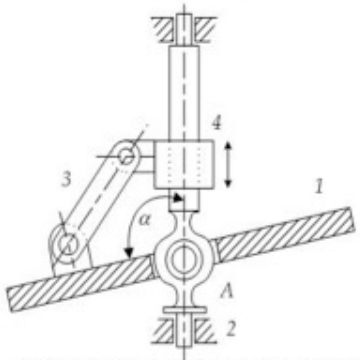


Marcus Vitruvius Pollio
"De Architectura Libri Decem" 27 BC

mechanical engineering handbooks

"Mechanisms of Modern Engineering Design": Ivan Artobolevsky 1947

Slider Crank Mechanism of a Centrifugal Governor
1636 SC:G

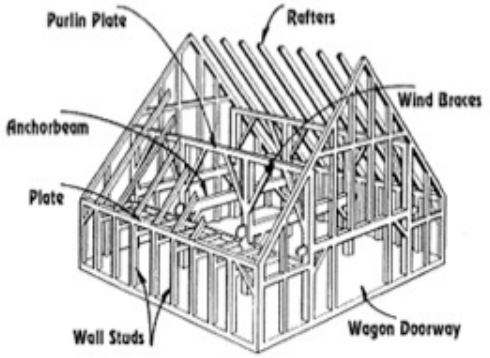


Link *1* is designed as a round plate turning about axis *A*. The angle α between the plane and the axis of rotation of shaft 2 depends upon the centrifugal force. When angle α is changed, connecting rod *3* slides sleeve 4 along the axis of shaft 2.

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

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*

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 ?

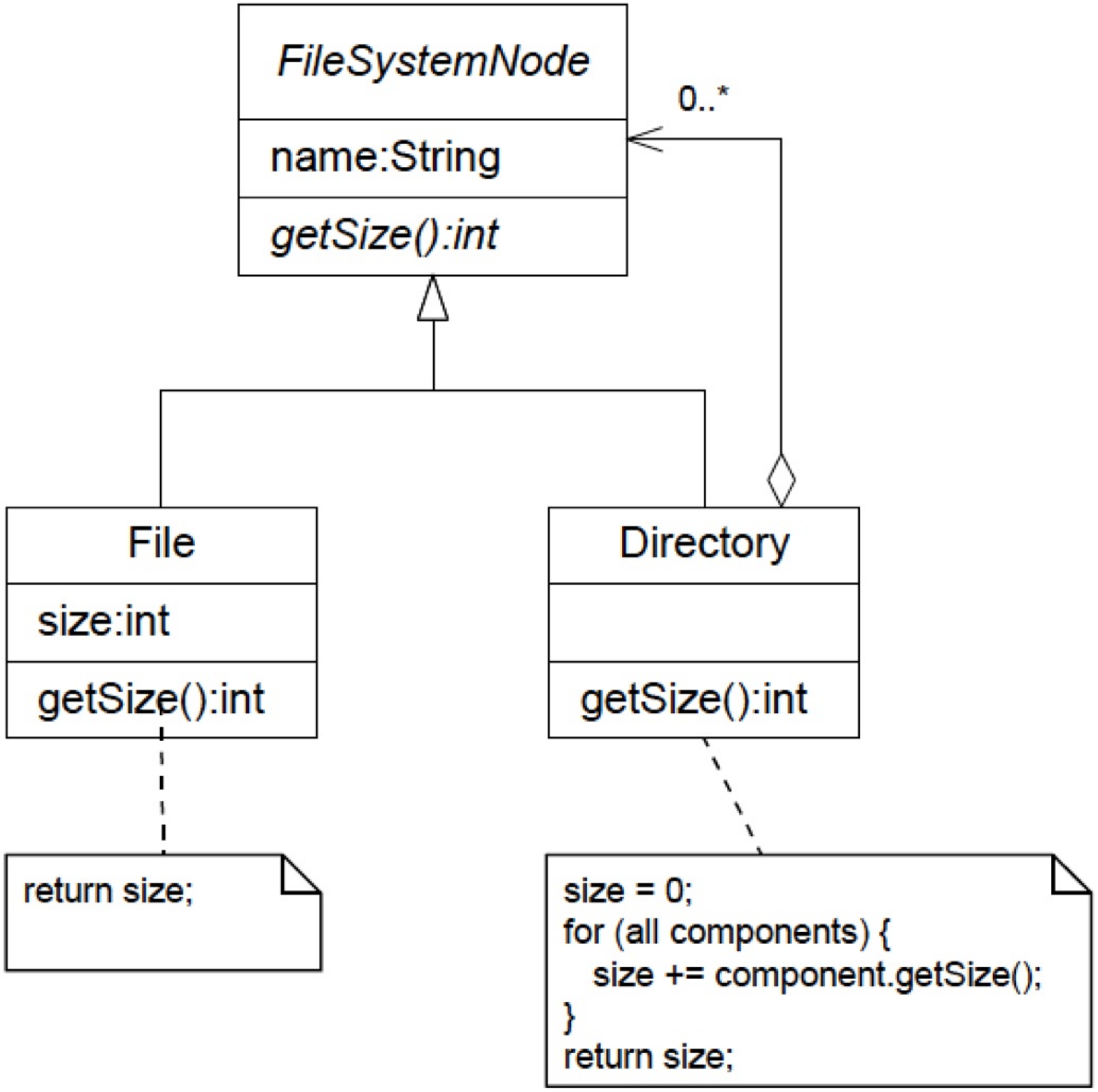
- Concept proposé pour les architectures de bâtiments (Christopher Alexander)
- Début d'application aux architectures logicielles en 1987
- Visibilité publique en 1994 grâce au livre
  - Design patterns: elements of reusable object-oriented software (Gamma, Helm, Johnson et Vlissides, dit le Gang of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

DIY

- Context: File System
  - Files, directories
- Goal: Compute the size 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*

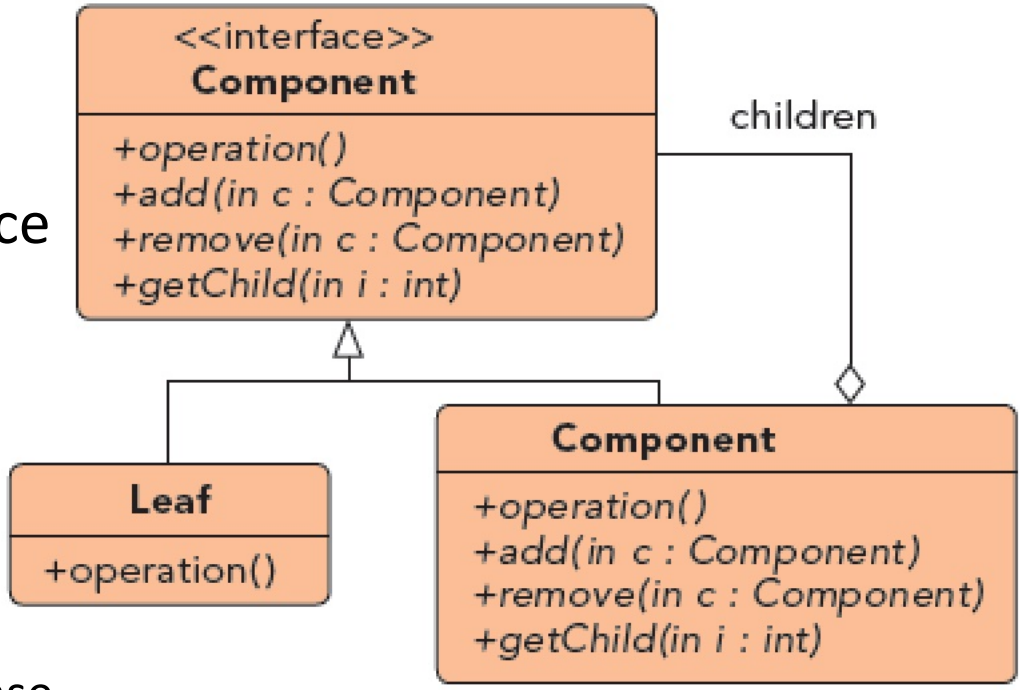
Design Pattern : Catégories ?

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight (195) Observer State Strategy Visitor

Composite

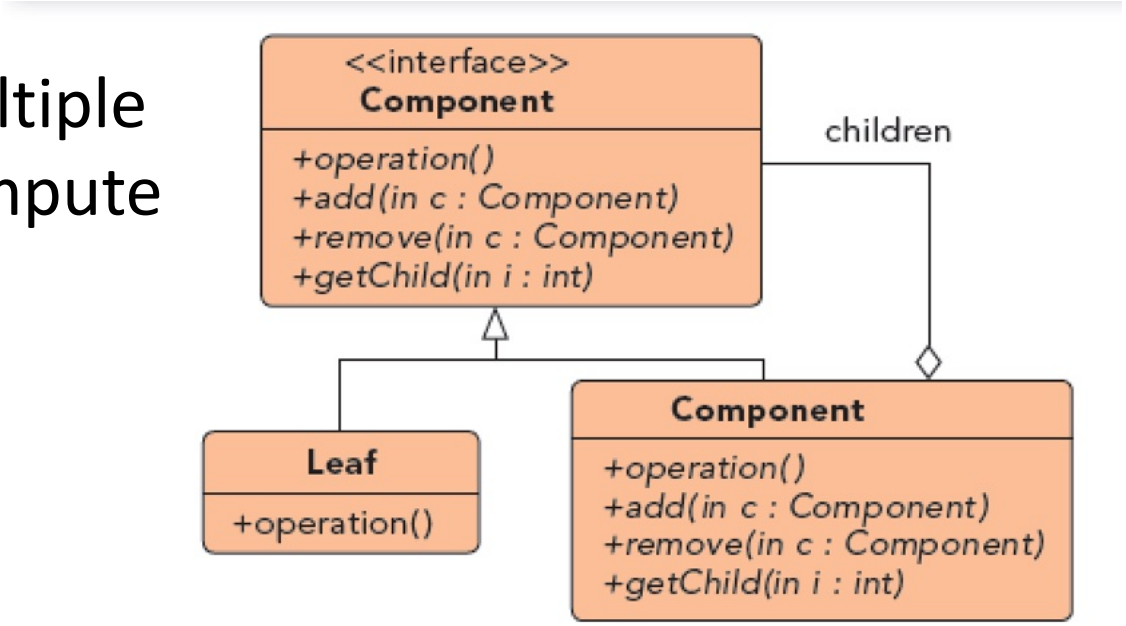
Composite

- 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

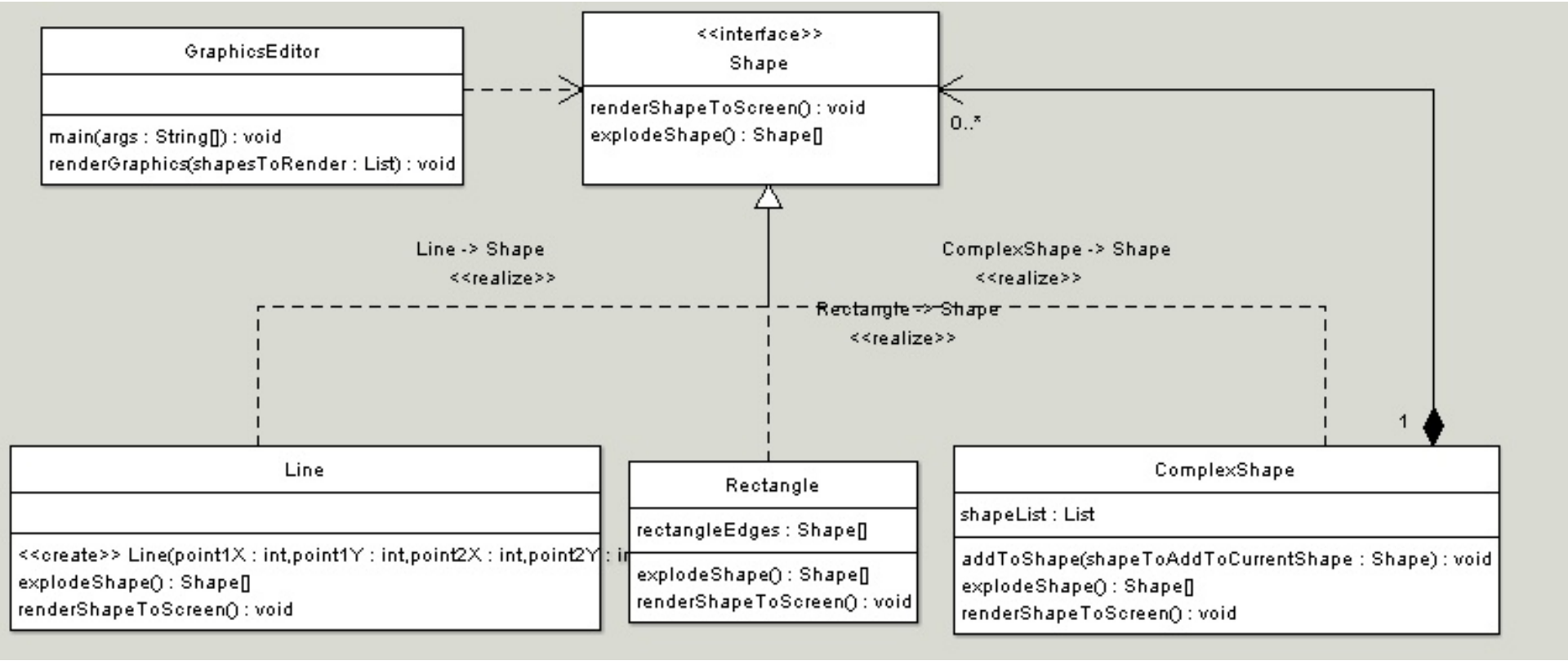


Composite

- 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)



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) Registers a new virtual-machine shutdown hook.
int	**availableProcessors**() Returns the number of processors available to the Java virtual machine.
Process	**exec**(String command) Executes the specified string command in a separate process.
Process	**exec**(String[] cmdarray) Executes the specified command and arguments in a separate process.
Process	**exec**(String[] cmdarray, String[] envp) Executes the specified command and arguments in a separate process with the specified environment.
Process	**exec**(String[] cmdarray, String[] envp, File dir) Executes the specified command and arguments in a separate process with the specified environment and working directory.
Process	**exec**(String command, String[] envp) Executes the specified string command in a separate process with the specified environment.
Process	**exec**(String command, String[] envp, File dir) Executes the specified string command in a separate process with the specified environment and working directory.
void	**exit**(int status) 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.

« 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() {
        // ...
    }
}
```

« Singleton »: Exercice

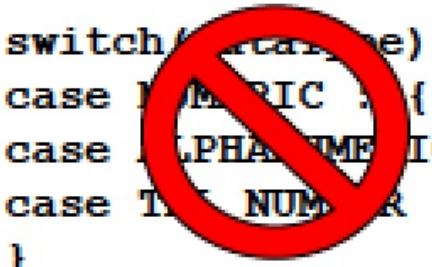
DIY

```
public class Q1 {
    public URL toURL(String path) {
        URL url;
        try { url = new URL(path); }
        catch(MalformedURLException e) {
            url = null;
        }
        return url;
    }
}
```

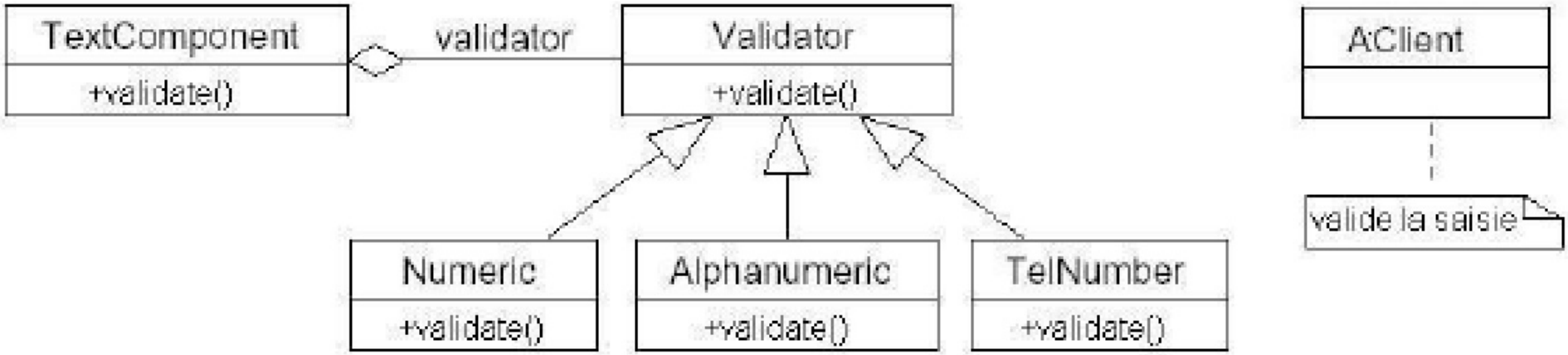
Lors de la conversion du String en URL , des erreurs peuvent survenir (MalformedURLException). On voudrait collecter ces erreurs dans un collecteur global à tout le programme.

Strategy

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



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



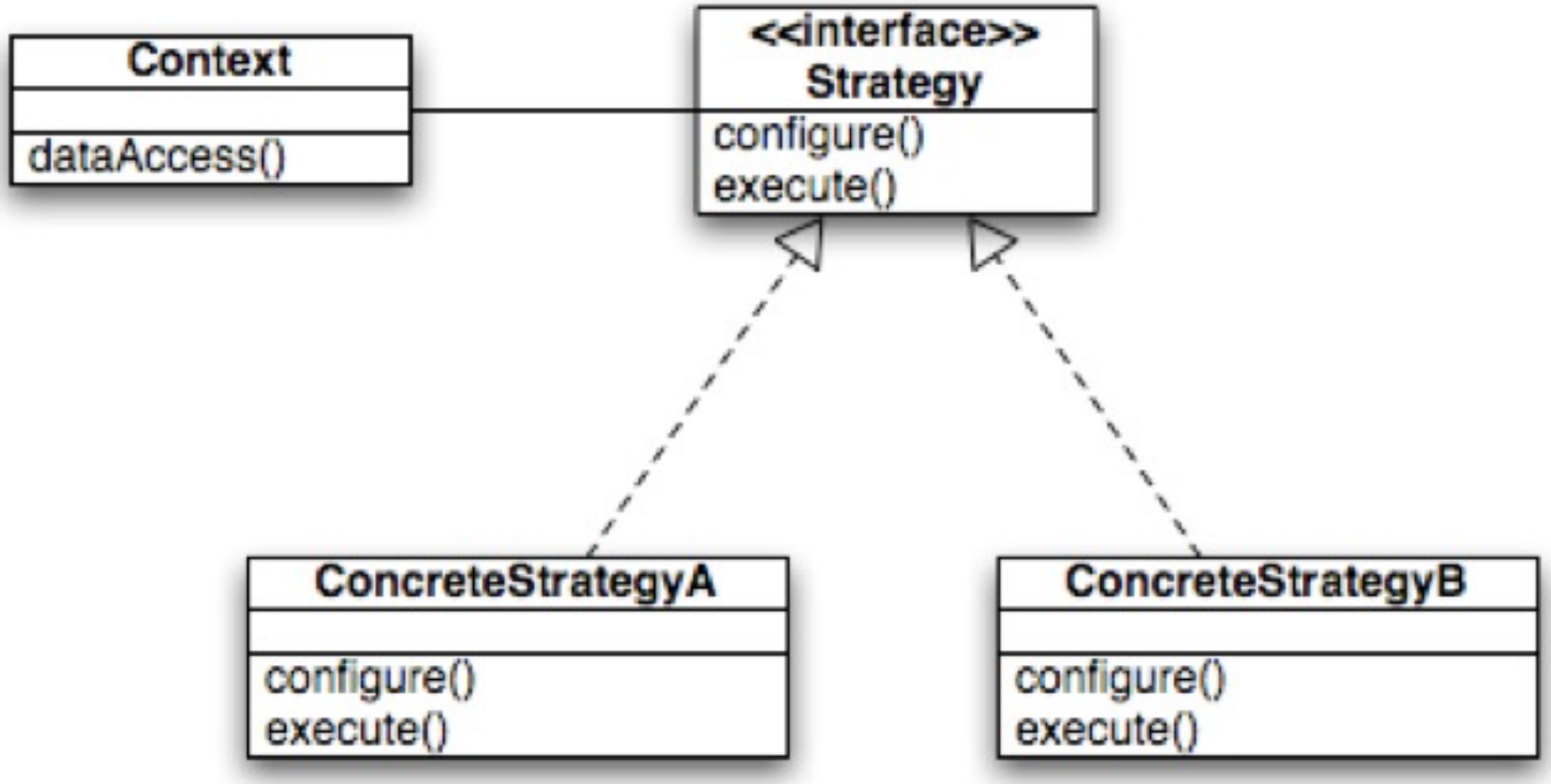
Patron « Strategy »

- 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 »

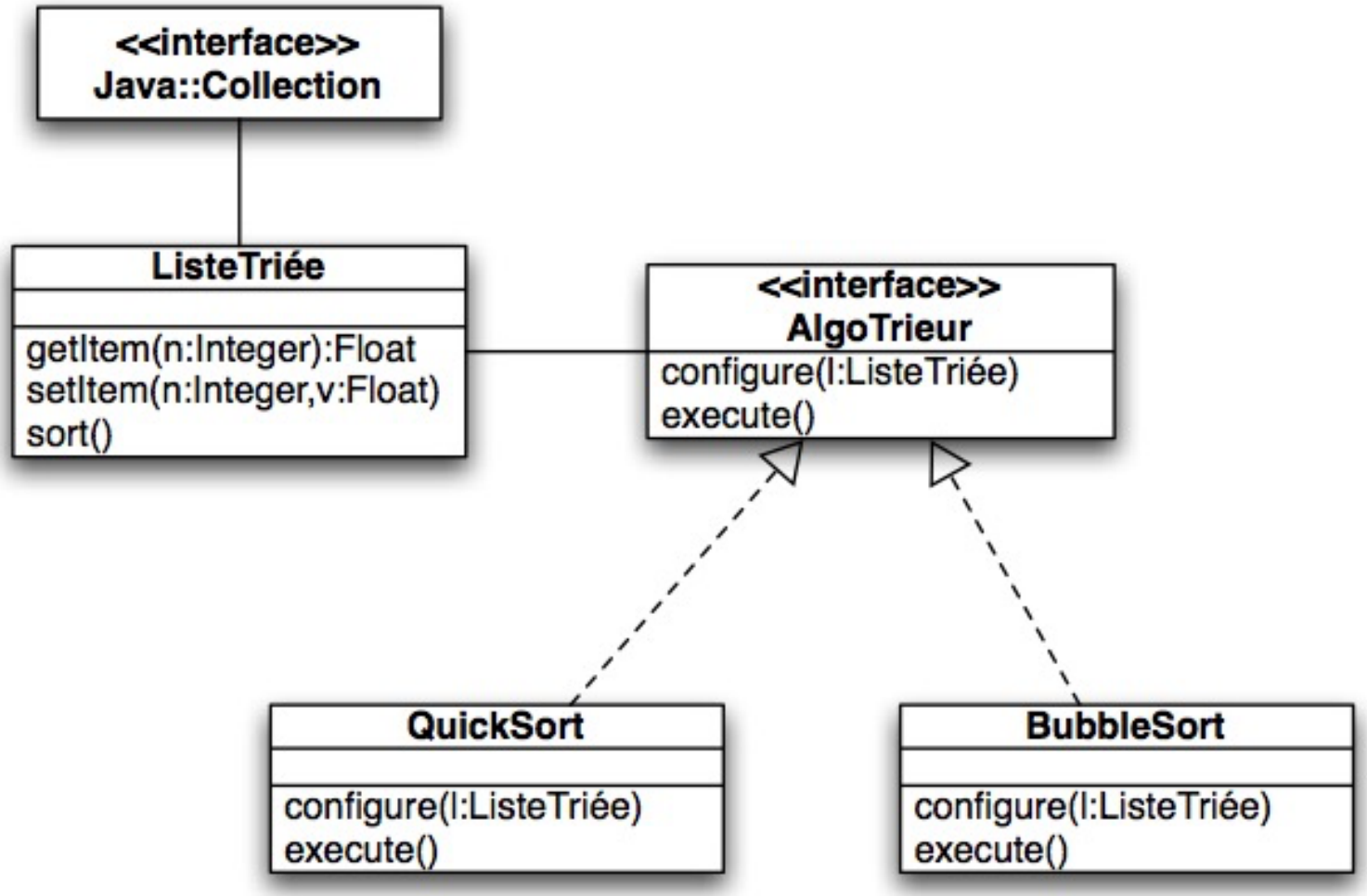
Patron « Strategy » - rôles

- 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



Patron « Strategy » - example



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

Strategy (behavior)

Comment choisir à l'exécution l'implémentation ?

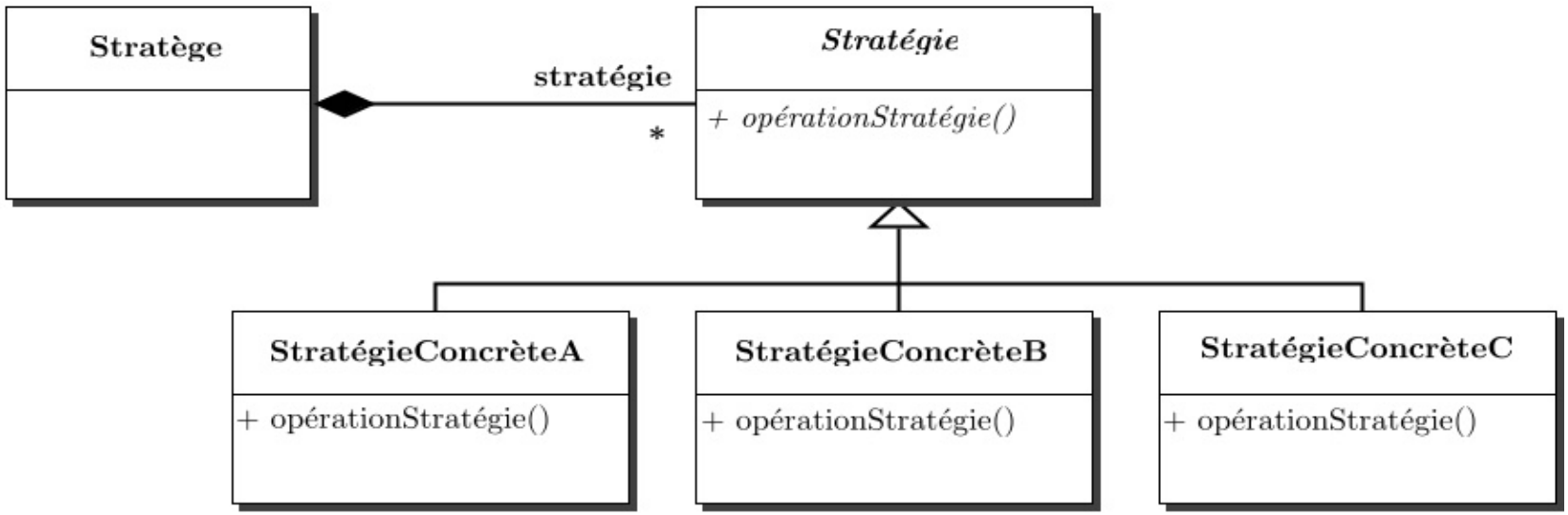
- On veut dans ce cas choisir en fonction du contexte telle ou telle **stratégie**

Strategy (behavior)

But :

Définir une famille d'algorithmes interchangeables dynamiquement

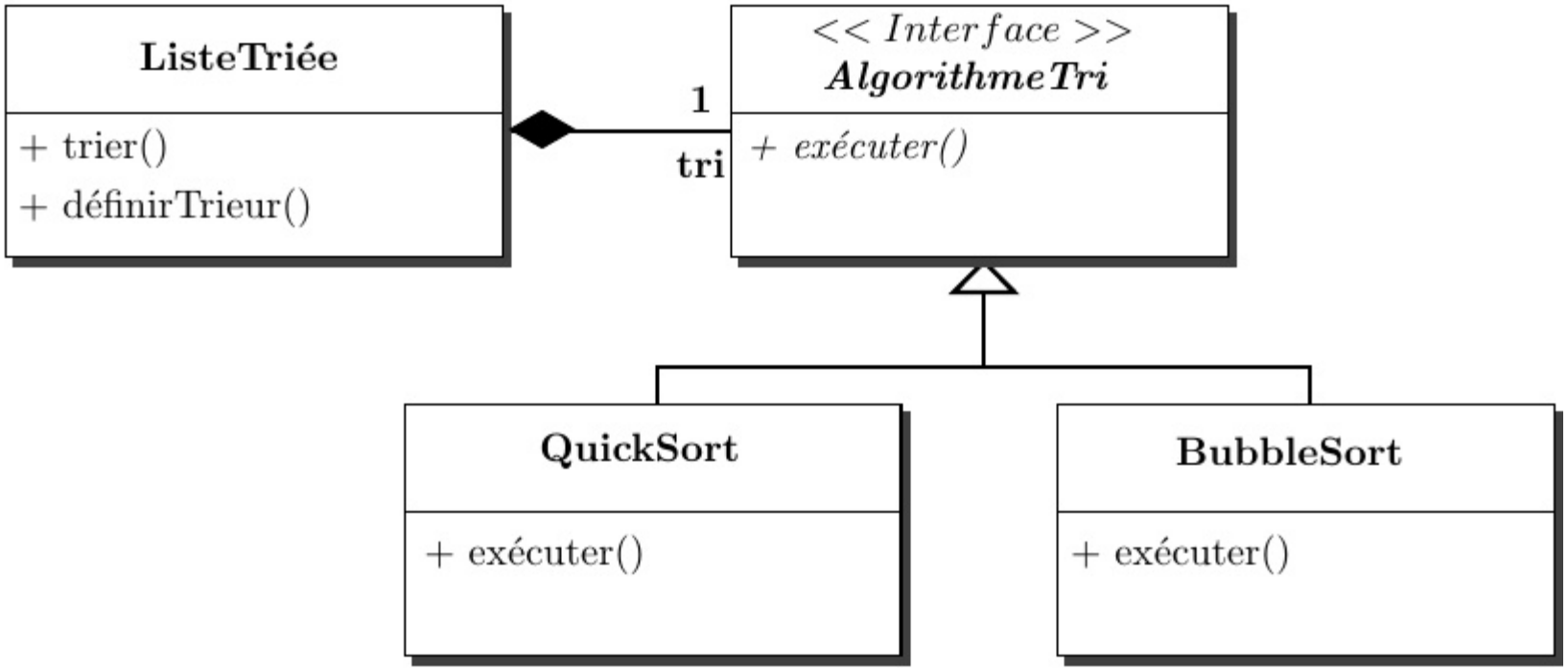
–**Stratégie** : interface pour 1 famille d'algorithmes
–**StratégieConcrète** : Implémentation d'un algorithme
–**Stratège**: désigne l'algorithme concret en vigueur,
peut fournir des données aux algorithmes



Strategy (behavior)

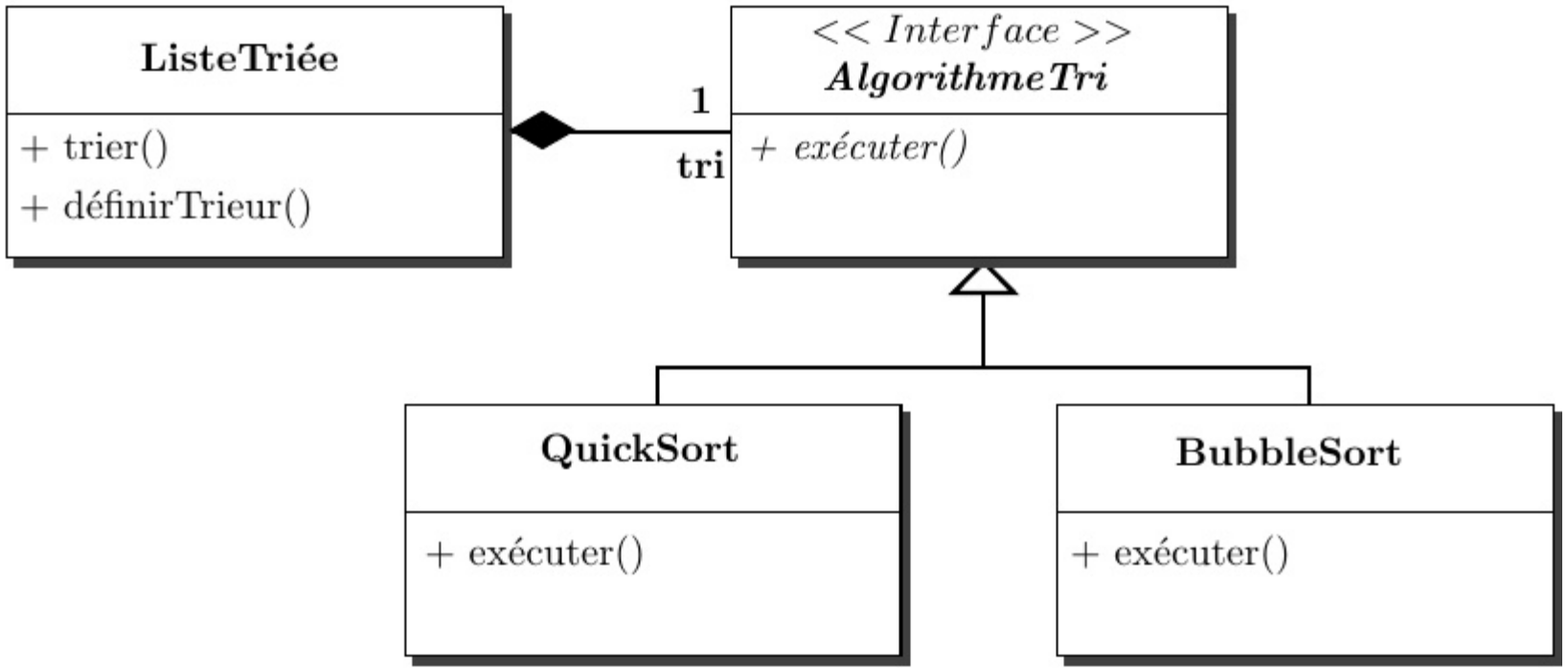
Exemple basique : liste triée

L'algorithme de tri peut être remplacé



Patron de Conception
Stratégie / Strategy *(comportement)*

```
ListeTriée liste = new ListeTriée() ;
liste.définirTrieur(new BubbleSort()) ;
// ...
Liste.trier() ;
// ...
liste.définirTrieur(new QuickSort()) ;
```



DIY

Patron « Strategy » - exercice

Un jeu vidéo peut avoir une partie en cours. Une partie possède un unique niveau de difficulté choisi lors de la création de la partie et pouvant changer au cours de celle-ci. Il existe 3 niveaux de difficulté : facile, normal et difficile.

Modéliser le jeu, les parties et les niveaux de difficultés en fonction du patron de conception *Stratégie*.

State

State

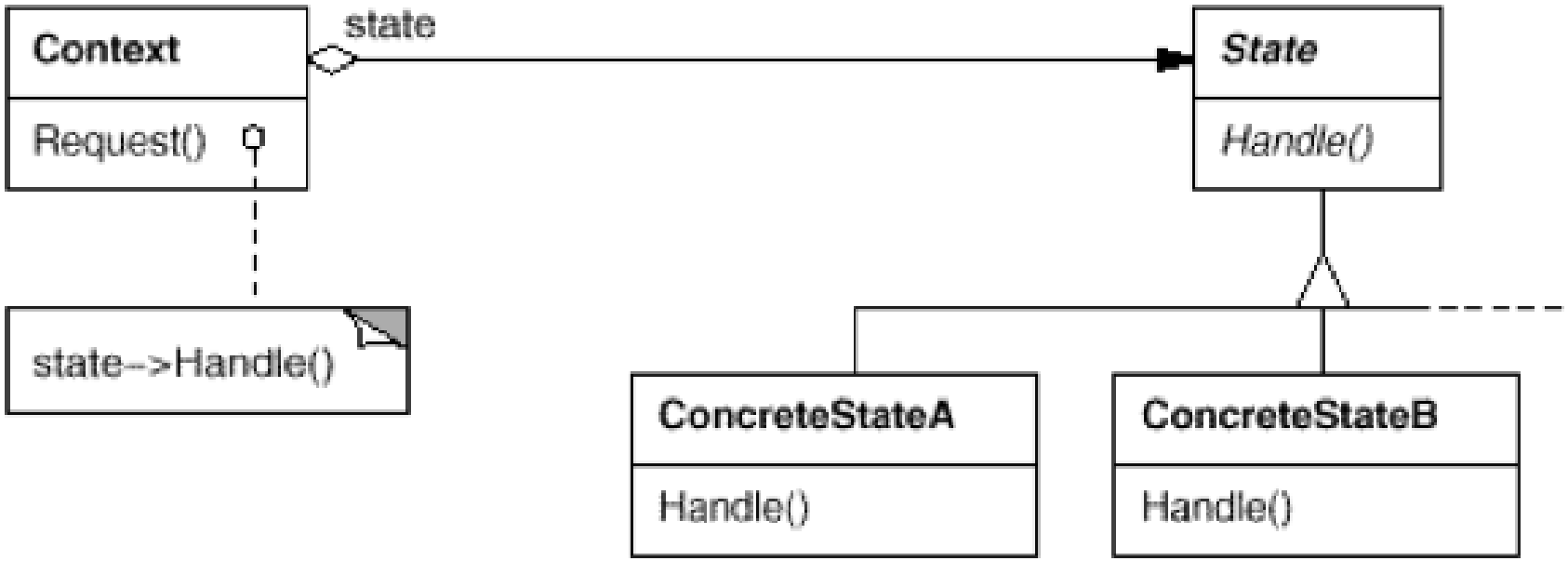
- Exemple
  - Éviter les instructions conditionnelles de grande taille (if then else)
  - Simplifier l'ajout et la suppression d'un état et le comportement qui lui est associé
  - Jeu vidéo: comportement des « bébêtes » avec mémoire de la position, de la vitesse et de la direction (intelligence) => décision de l'évolution de l'état
  - État d'une 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

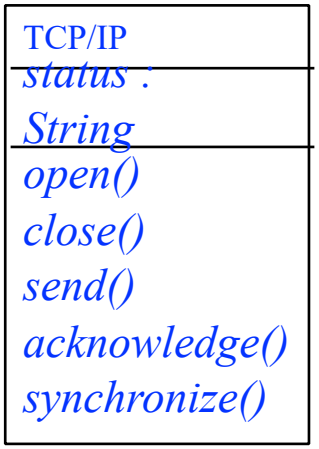
Patron « State » - structure

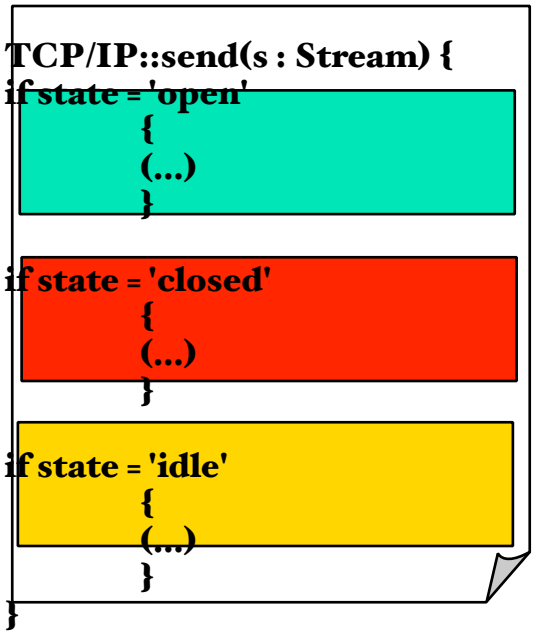


Context est une classe qui permet d'utiliser un objet à état et qui gère une instance d'un objet ConcreteState
State définit une interface qui encapsule le comportement associé avec un état particulier de Context
Les ConcretState implémentent un comportement associé avec l'état de Context

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

Implémentation de TCP/IP





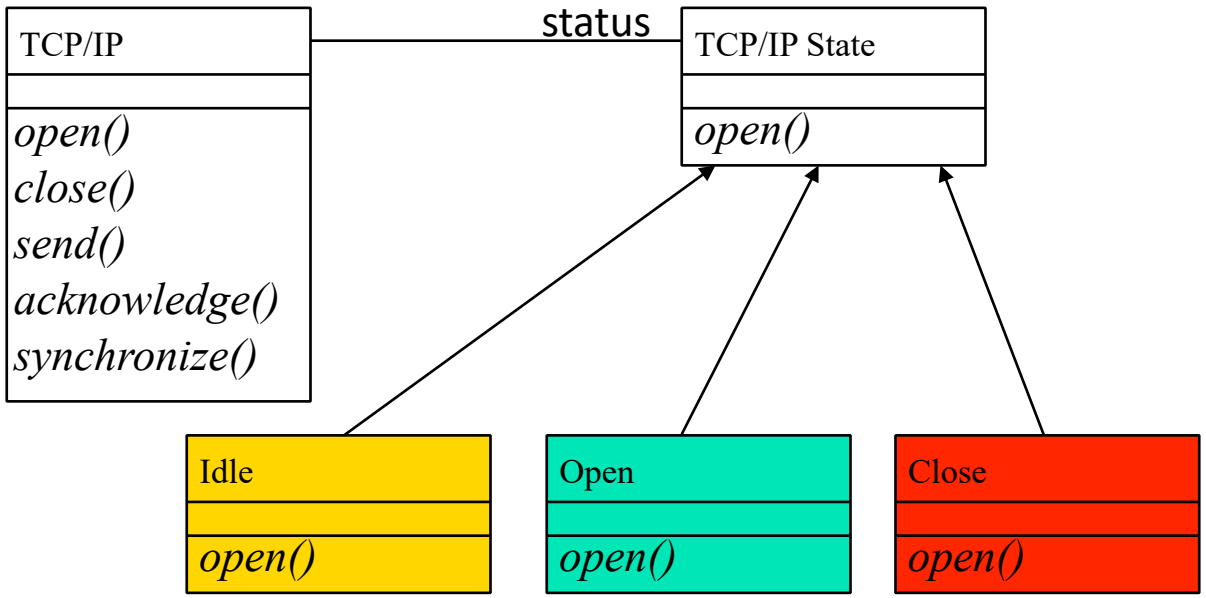
Problème

- Comment éviter que l'état de la connexion soit vérifié à chaque fois qu'un paquet est envoyé?

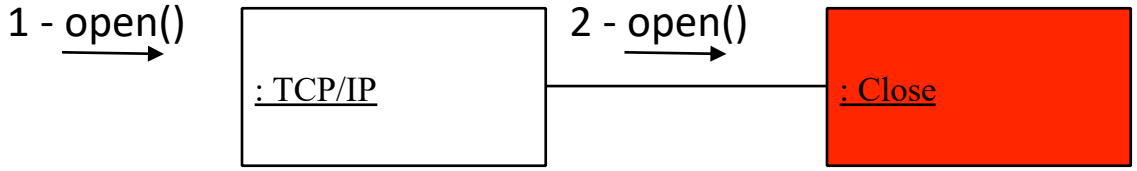
Solution

- Isoler les comportements dépendants des différents états de connexion dans des classes différentes.

En d'autres termes:

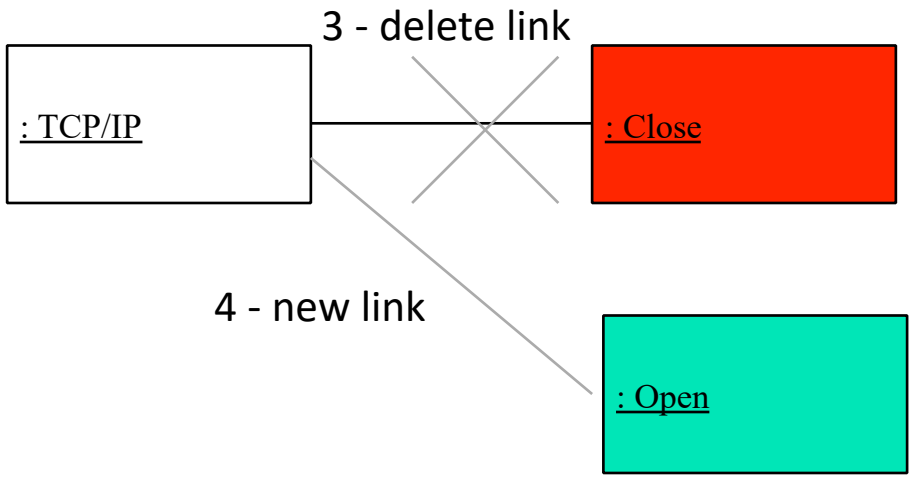


Exemple de connexion




```
TCP/IP::open() {
        this.status.open();
}
```

Exemple de connexion



Conséquences

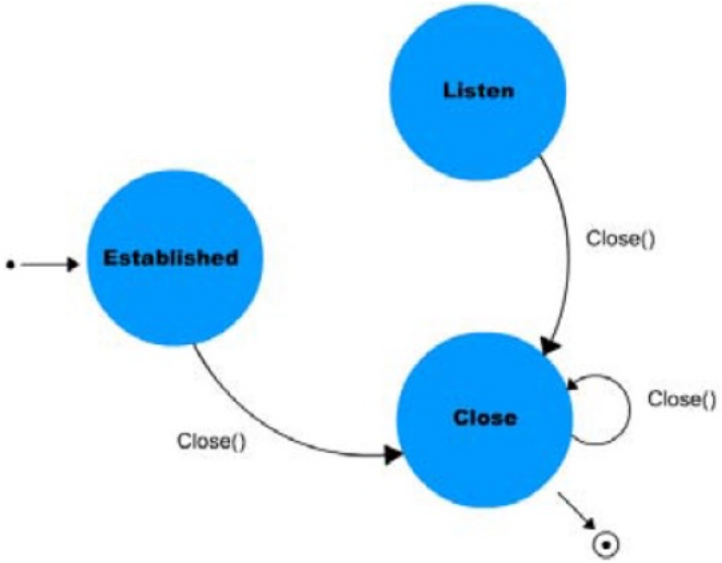
- Chaque instance de la classe “TCP/IP” est associée à une instance d’une sous-classe de “TCP/IP State”.
- La vérification de l’état n’est plus nécessaire.

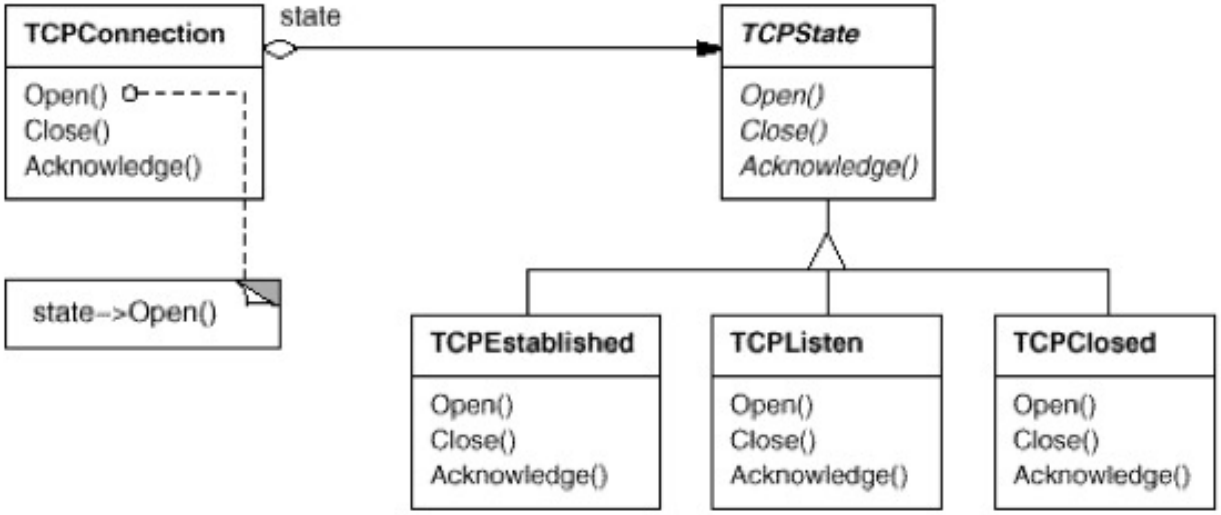
Observations

- Cette solution est utilisée dans la plupart des implémentations du protocole TCP/IP sur Unix.

- Une solution similaire est utilisée dans plusieurs éditeurs graphiques (comportement d'un outil selon le type de figure sélectionnée).

Patron « State » - example





Conséquences

- Le comportement spécifique à un état est isolé.
- Les transitions d'état sont explicites.
- Les objets-états peuvent être partagés.

Compromis d'implémentation

- Qui définit les transitions d'état?
- Création et destruction des états.
- Utilisation du dispatch dynamique.

Patron « State » - exercice

DIY

La montre digitale

La montre que nous étudions est très simple. Elle possède deux boutons : *avance* et *mode*. Le mode courant est le mode *Affichage*. Quand on appuie une fois sur le bouton *mode*, la montre passe en *modification heure*. Chaque pression sur le bouton *avance* incrémente l'heure d'une unité. Quand on appuie une nouvelle fois sur le bouton *mode*, la montre passe en *modification minute*. Chaque pression sur le bouton *avance* incrémente les minutes d'une unité. Quand on appuie une nouvelle fois sur le bouton *mode*, la montre repasse en mode *affichage*.

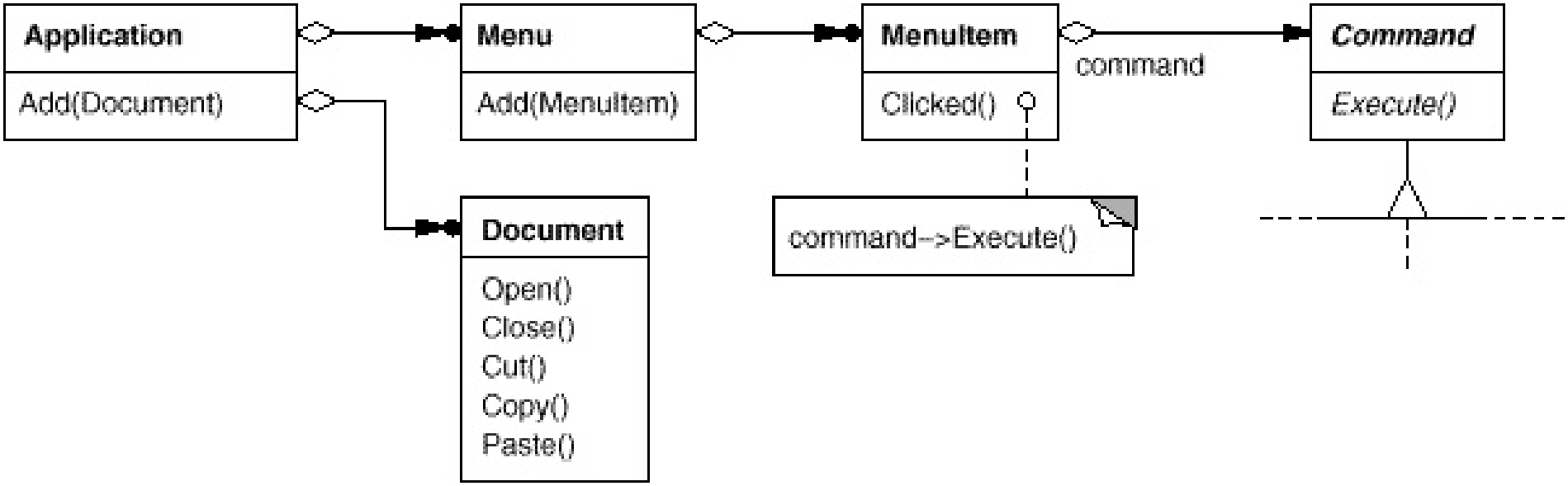
1. Représenter le diagramme d'états de la montre
2. Proposez un codage simple de la montre. Etudiez ensuite les modifications à apporter à votre version avec les modifications suivantes :
   a. Une nouvelle fonctionnalité apparaît dans la montre : le réglage des secondes (quand on presse le bouton *mode* en cours de réglage des minutes, le réglage consiste juste en une remise à zéro en appuyant sur *avance*).
   b. Un nouveau bouton apparait : l'*avance rapide*, qui permet d'incrémenter les heures de 5 en 5, et les minutes de 10 en 10.
   c. Une nouvelle fonctionnalité apparait : quand des actions sont sans effet (p.ex., *avance* ou *avance rapide* en mode *affichage*), la montre émet un bip.
3. Analysez votre solution du point de vue de la maintenance.
4. Proposez une solution en utilisant cette fois le patron de conception *State*. Reprenez les modifications de la question 2.
5. Etudiez les différentes implémentations du design pattern *State*.

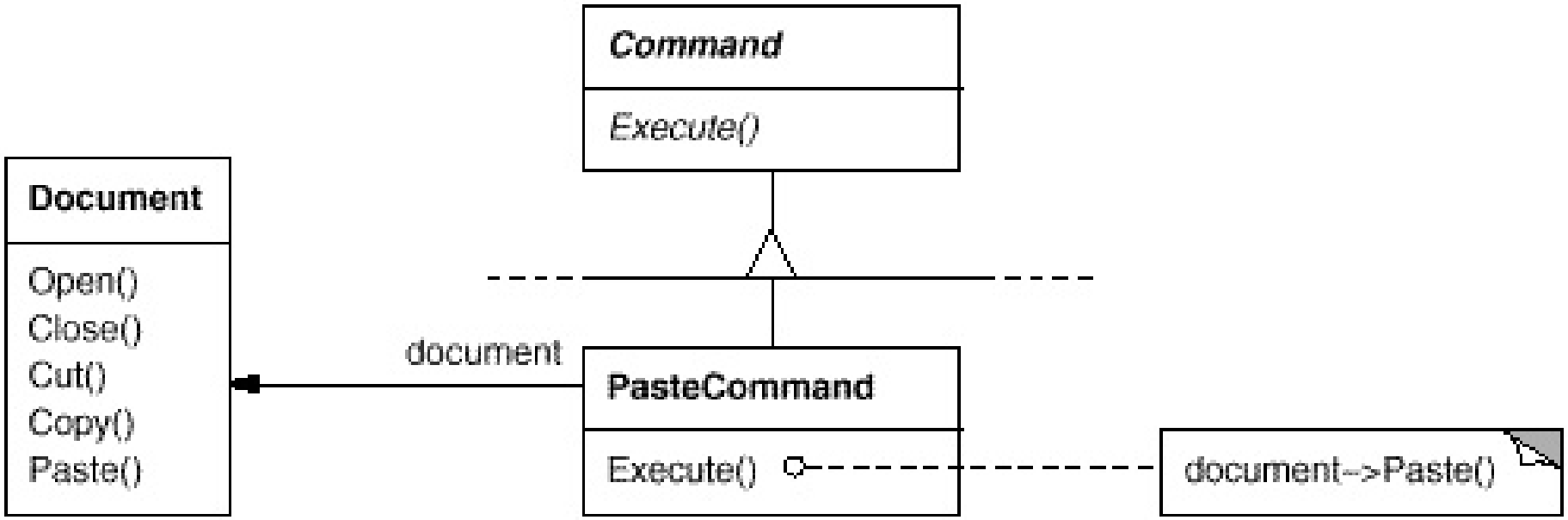
Command

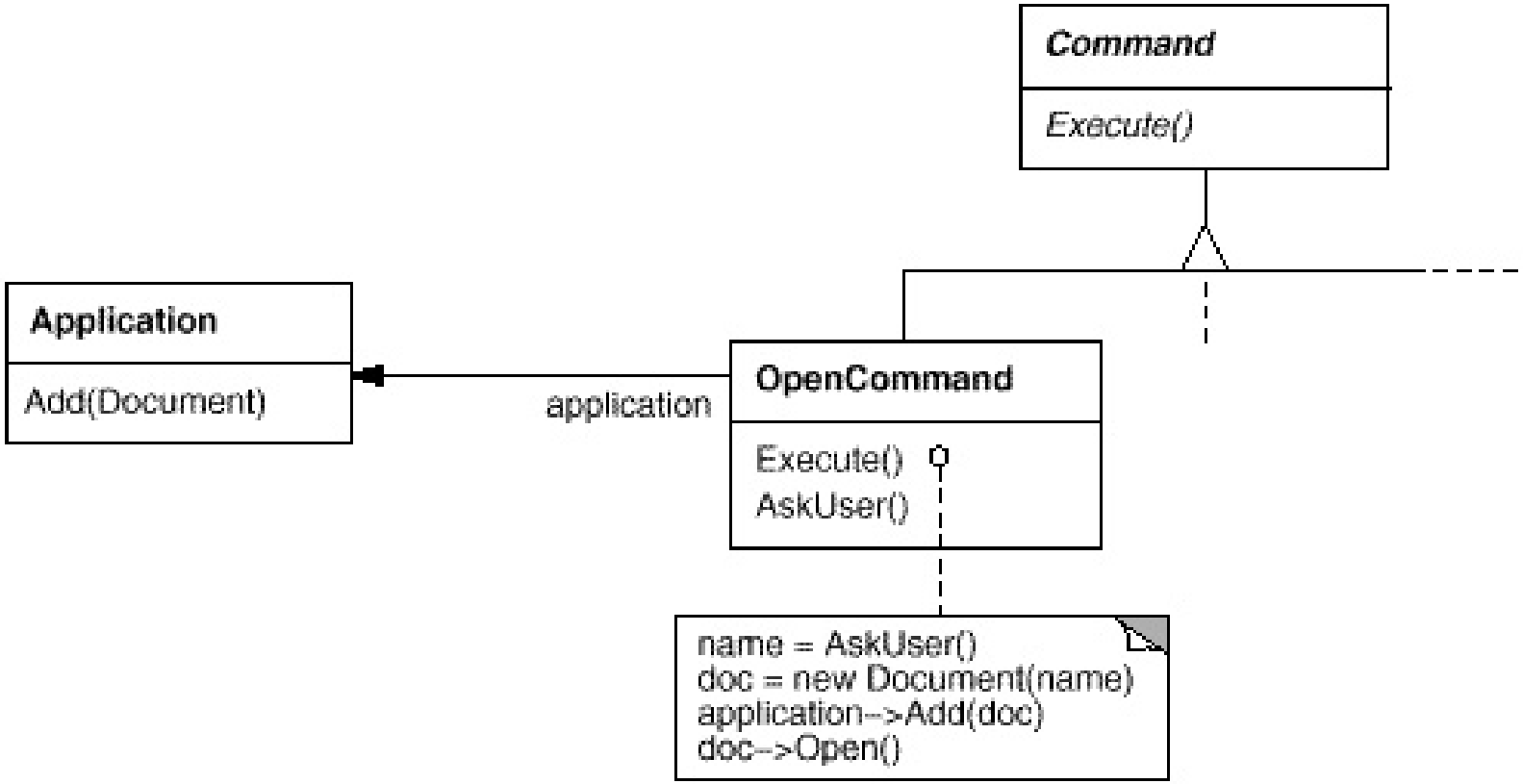
Patron « Command »

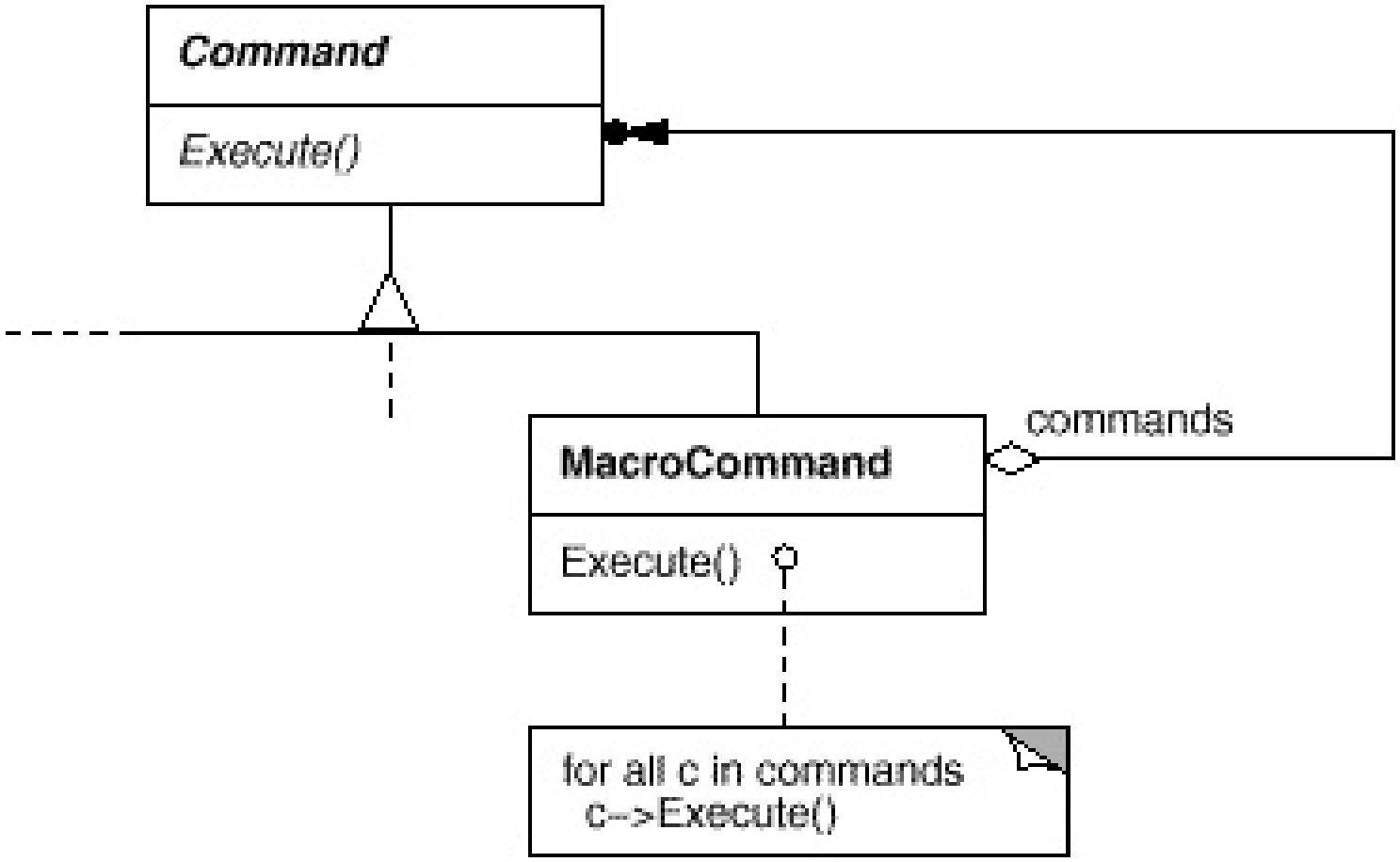
- L'objectif est de découpler
  - le choix d'une action à faire dans une certaine situation (e.g., undo)
  - la détection de la situation et l'exécution de l'action décidée

➔ 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()

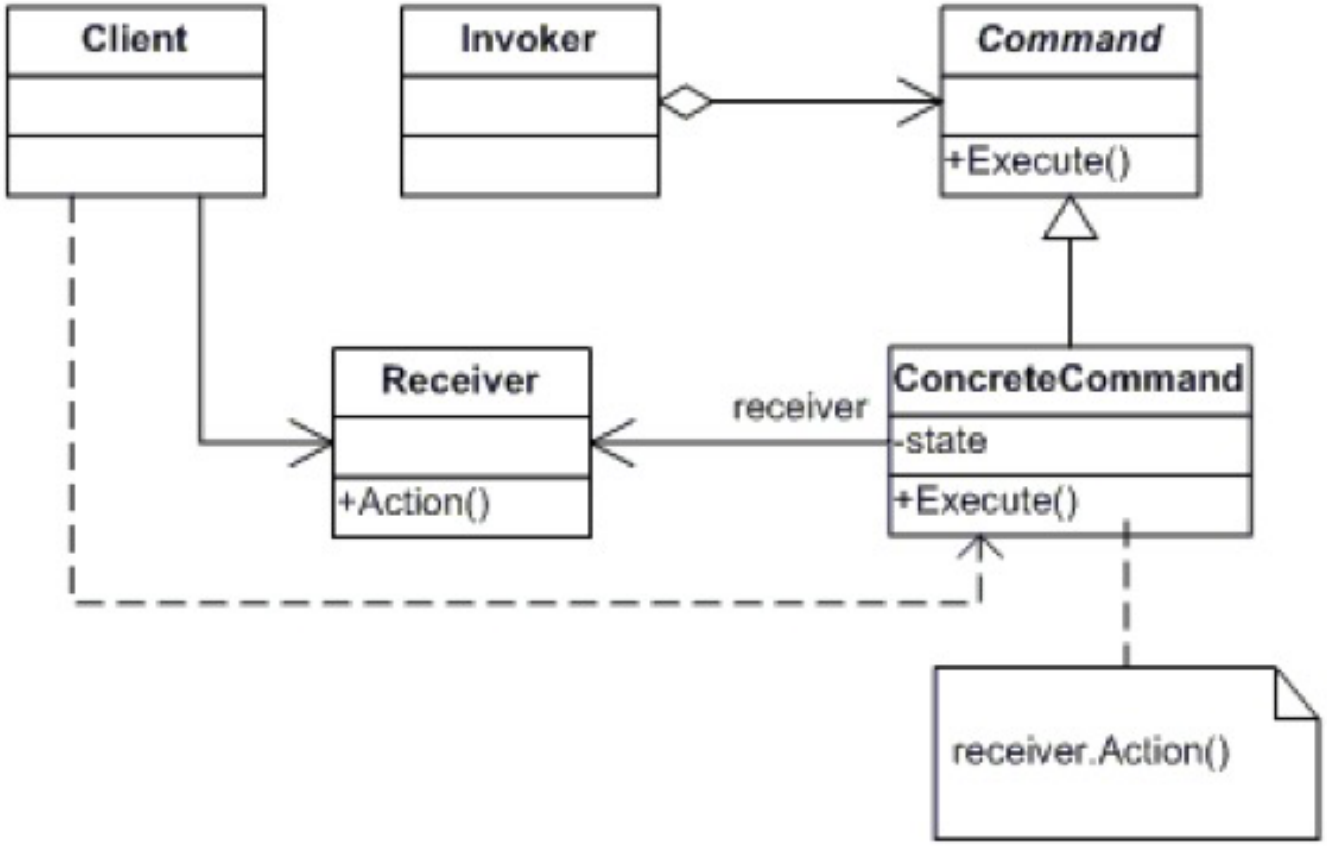

Command
Execute()
Application
Add(Document)
application
OpenCommand
Execute()
AskUser()
name = AskUser()
doc = new Document(name)
application->Add(doc)
doc->Open()


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

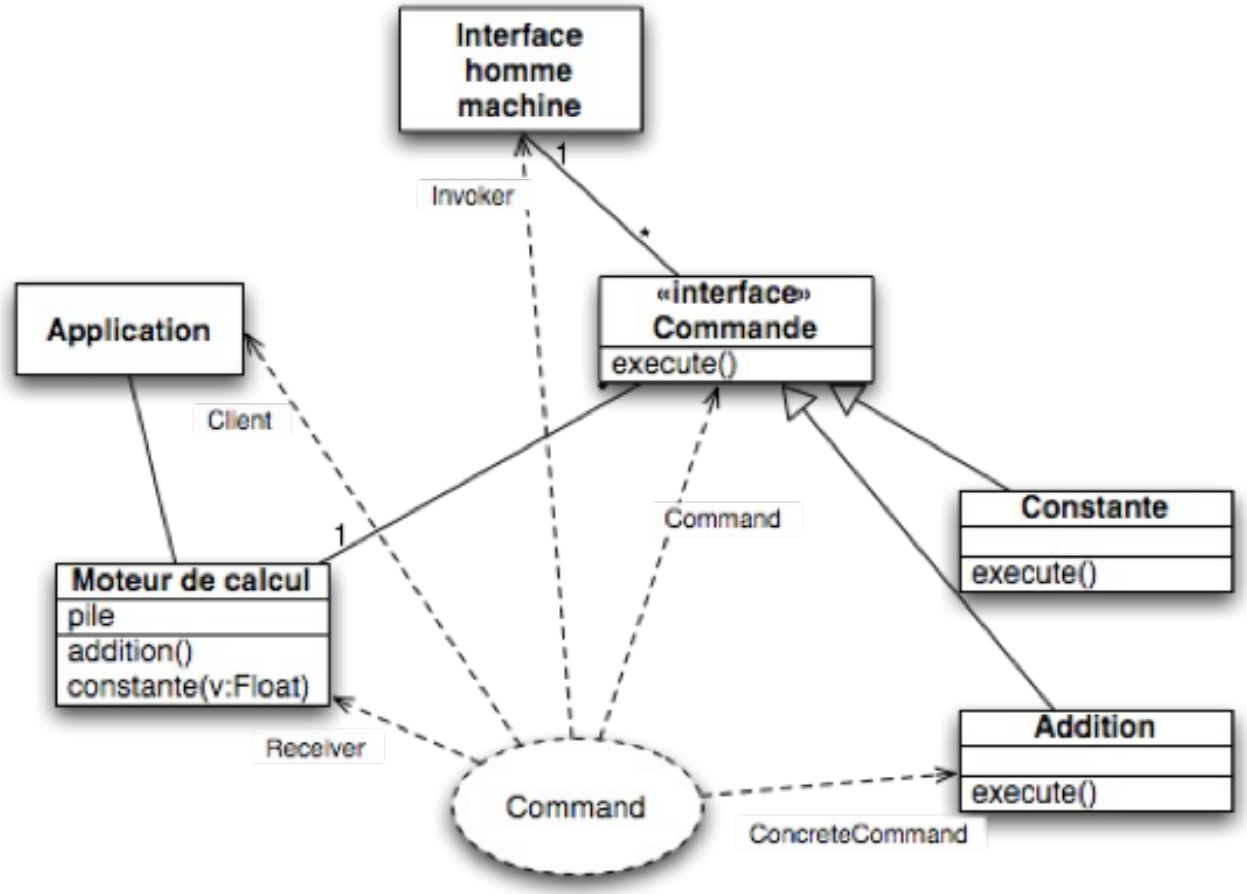
Patron « Command » - rôles

- 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



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 »)

Patron de Conception
Commande / Command *(comportement)*

Command in Java 8 with Lambdas

```
public class Macro {
 private final List<Command> cmds;

 public Macro() {
  cmds = new ArrayList<>();
 }

 public void addCommand(Command cmd) {
  System.out.println(cmd);
  cmds.add(cmd);
}

 public void run() {
  cmds.forEach(Command::execute);
 }
}
```

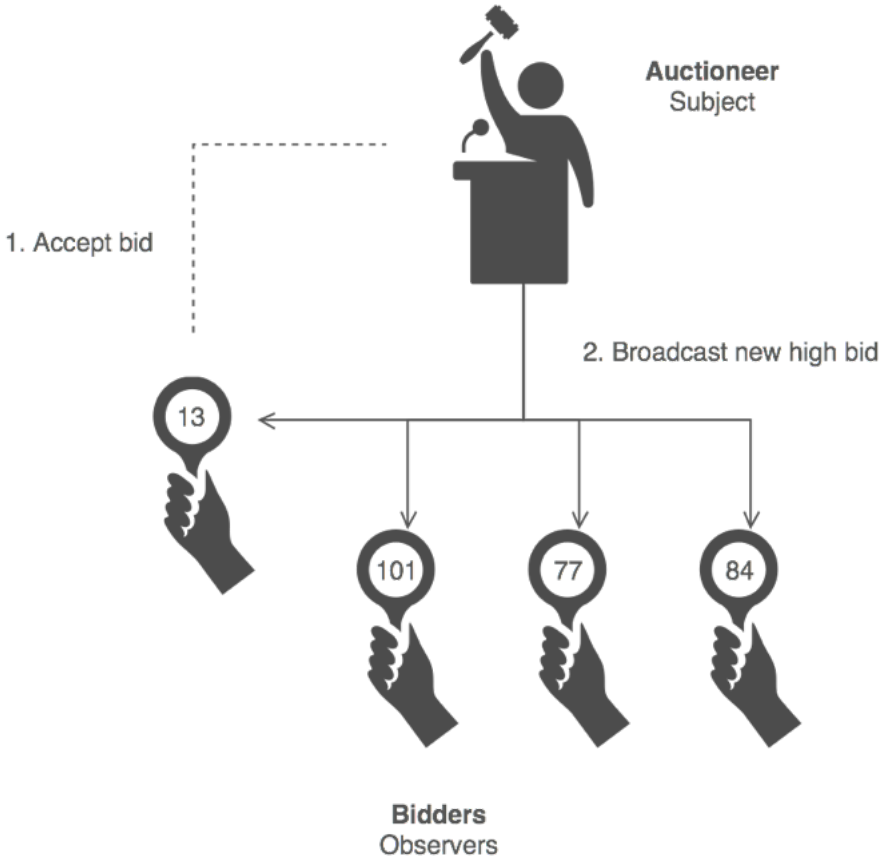
```
Output:
test.Main$1@15db9742
test.Main$$Lambda$1/2055281021@24d46ca6
test.Main$$Lambda$2/925858445@2f92e0f4
open
open
open
```

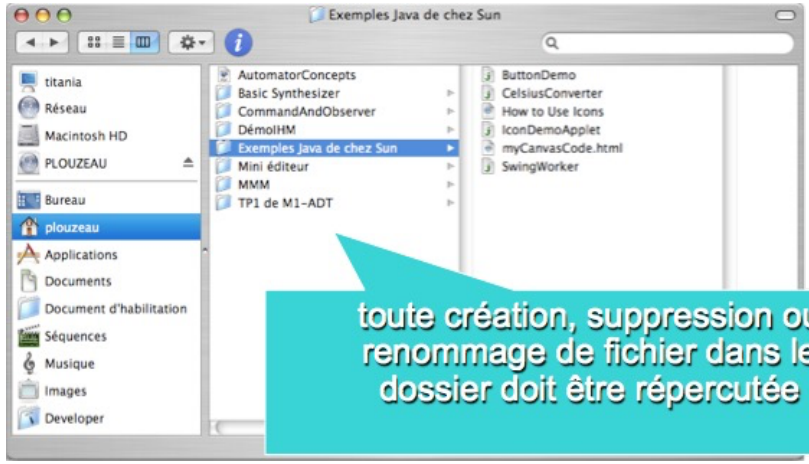
```
Macro macro = new Macro();
Editor editor = new EditorImpl();

// Java 7 version: using an
// anonymous class. Too much verbose
macro.addCommand(new Command() {
 @Override public void execute() {
  editor.open();
}});

// Java 8 brings lambdas
// (anonymous operations)
macro.addCommand(() -> editor.open());

// Idem than before but more concise
macro.addCommand(editor::open);

List<String> myList = new ArrayList<>();
// The normal syntax of the forEach lambda
myList.forEach((String elt) ->
              {System.out.println(elt);});
// Sugar syntax to shorten the lambda
myList.forEach(elt ->
              System.out.println(elt));

macro.run();
```

Observer

Patron « Observer »

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

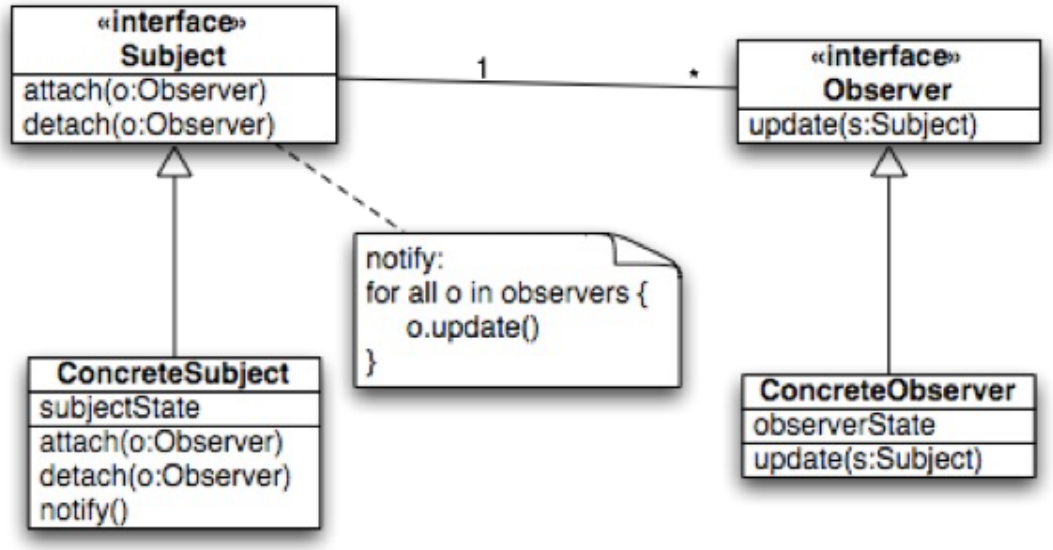




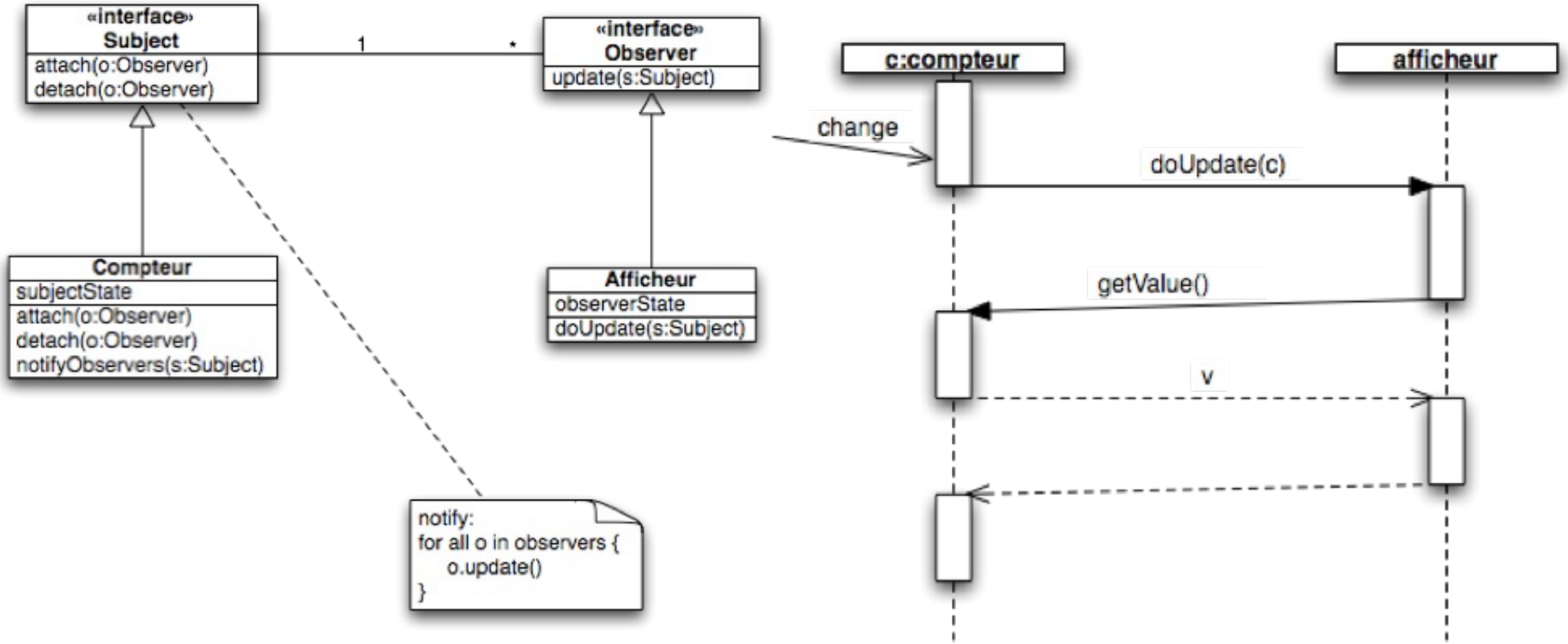
Patron « Observer » - rôles

- 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



Template Method

```
public class PlainTextDocument {

   ...

   public void printPage (Page page) {

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

   }

   ...

}
```

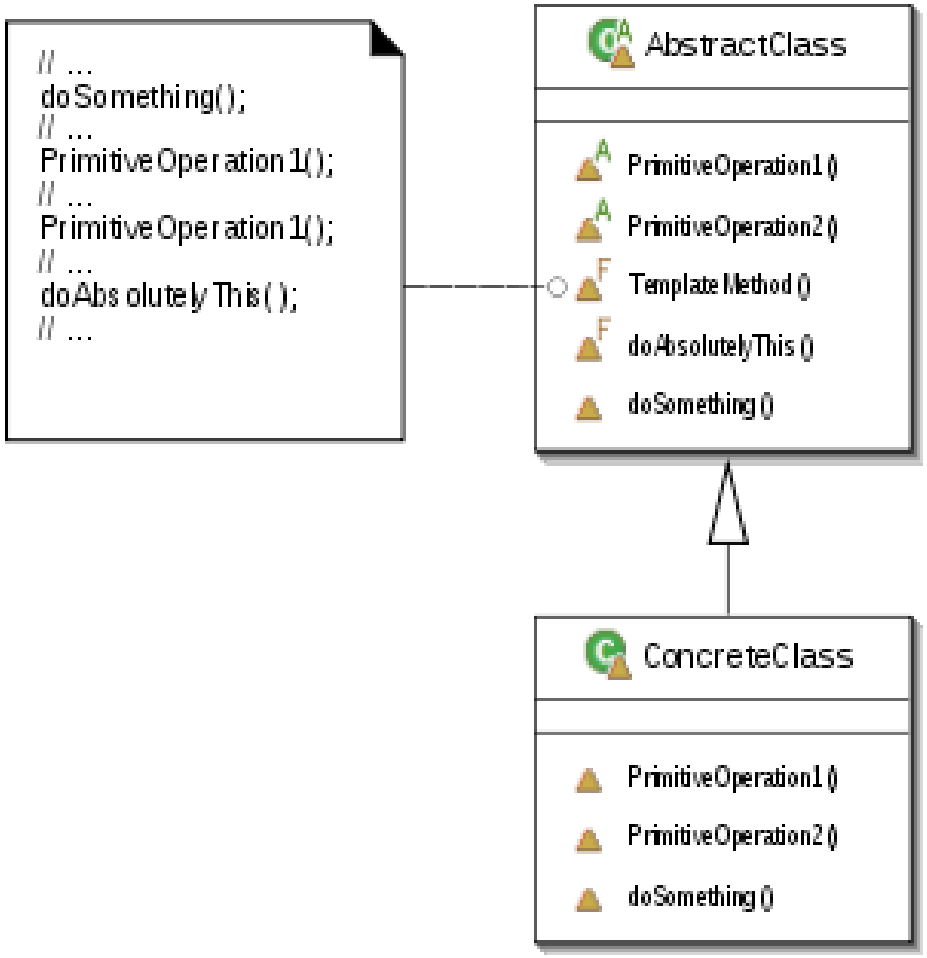
```
public class HtmlTextDocument {

   ...

   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() {}
   ...
}
```

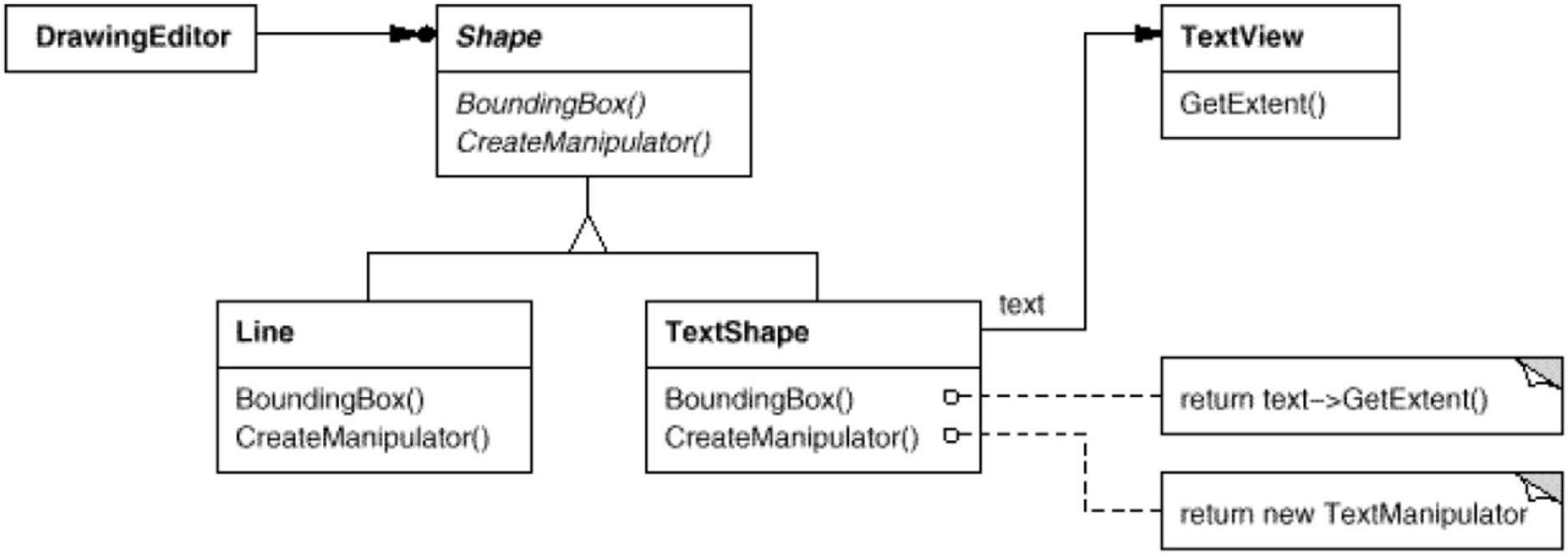
Template Method in an existing API

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

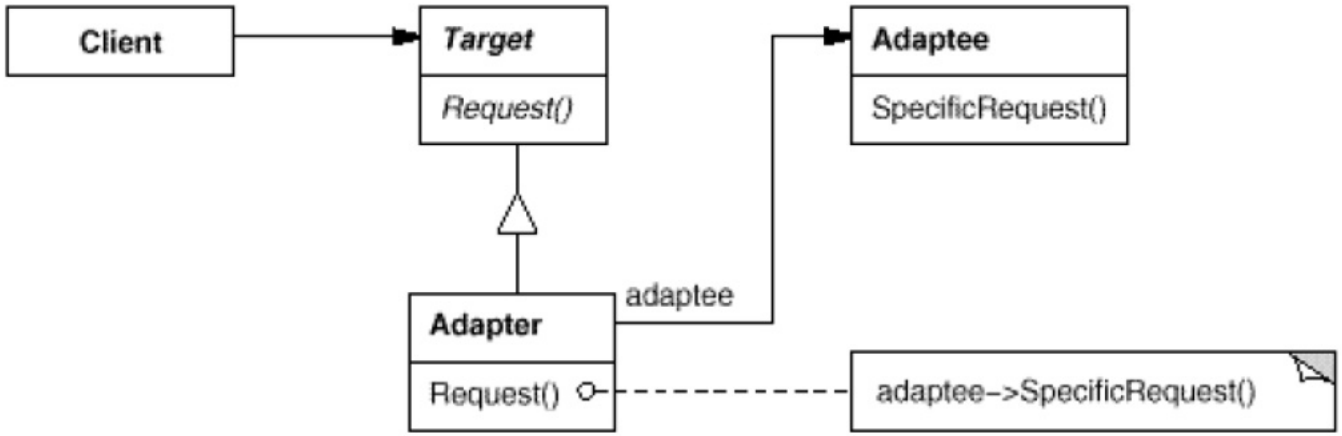
- Demo

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



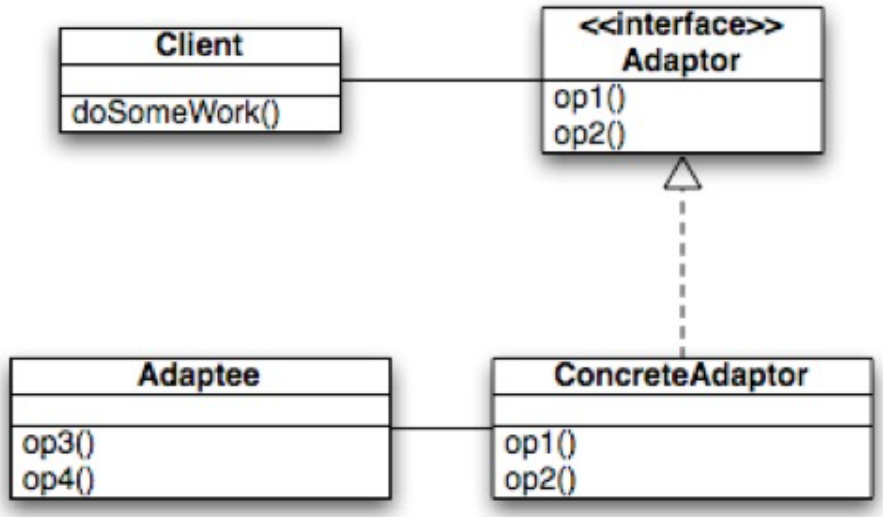
Patron « Adapter »

- 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

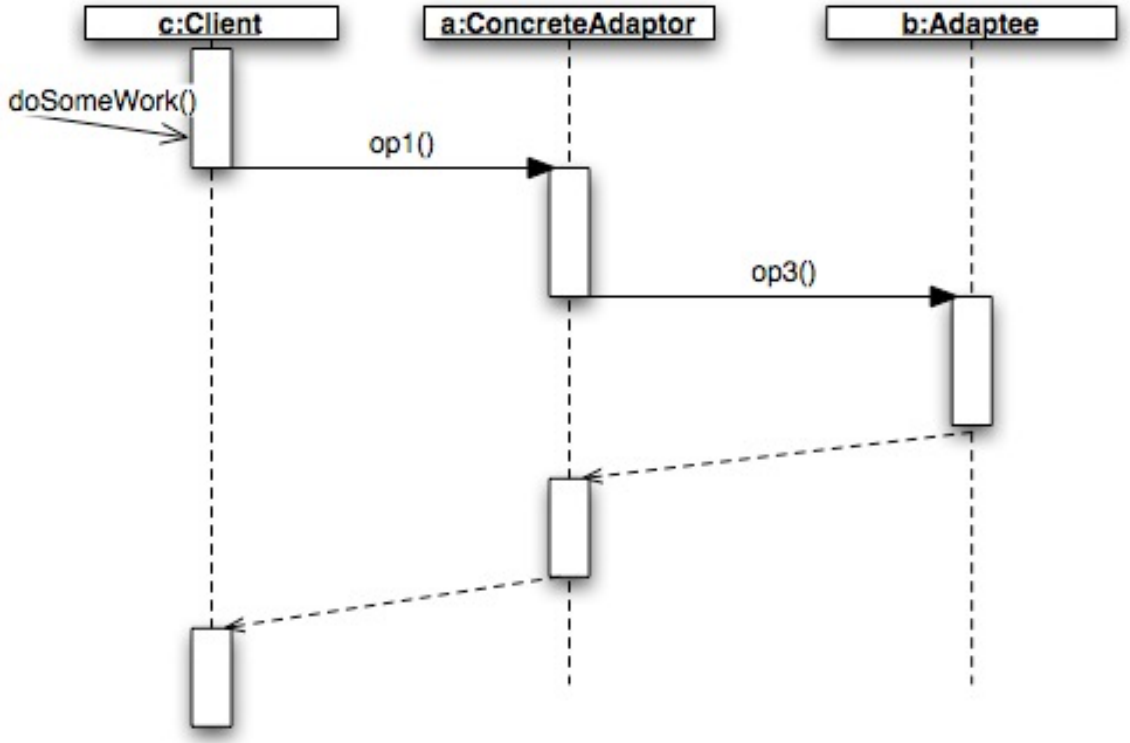
Patron « Adapter » - rôles

- 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

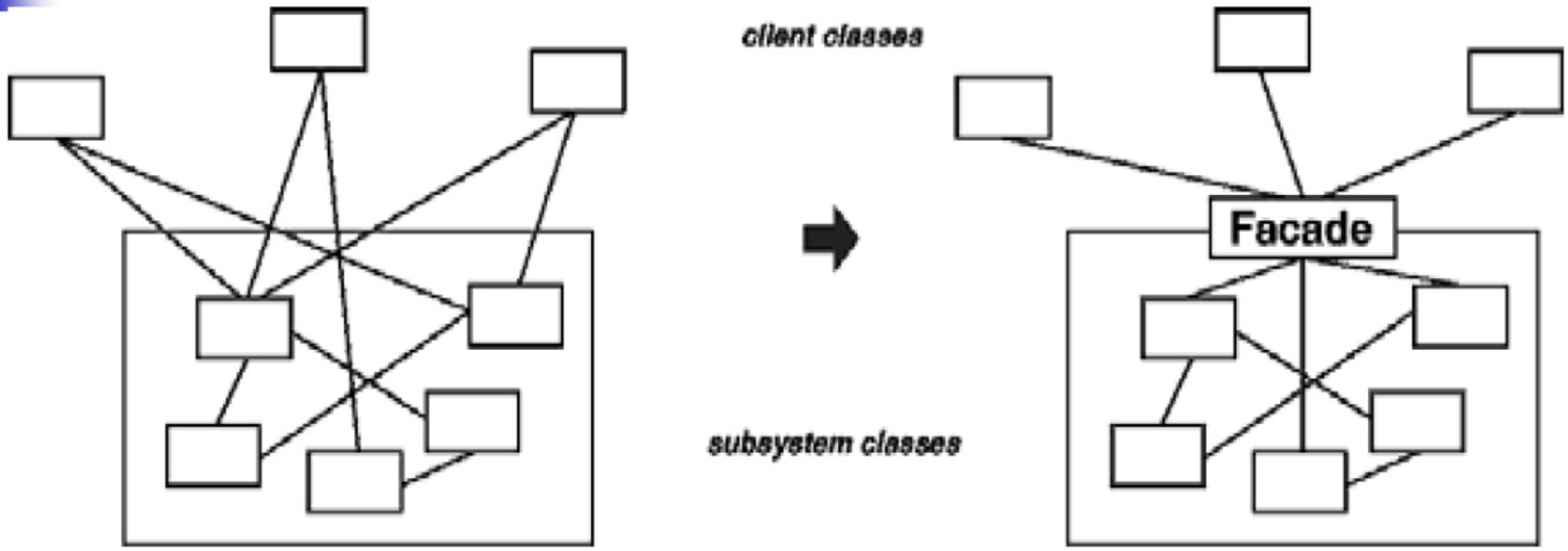
Patron « Adapter » - structure

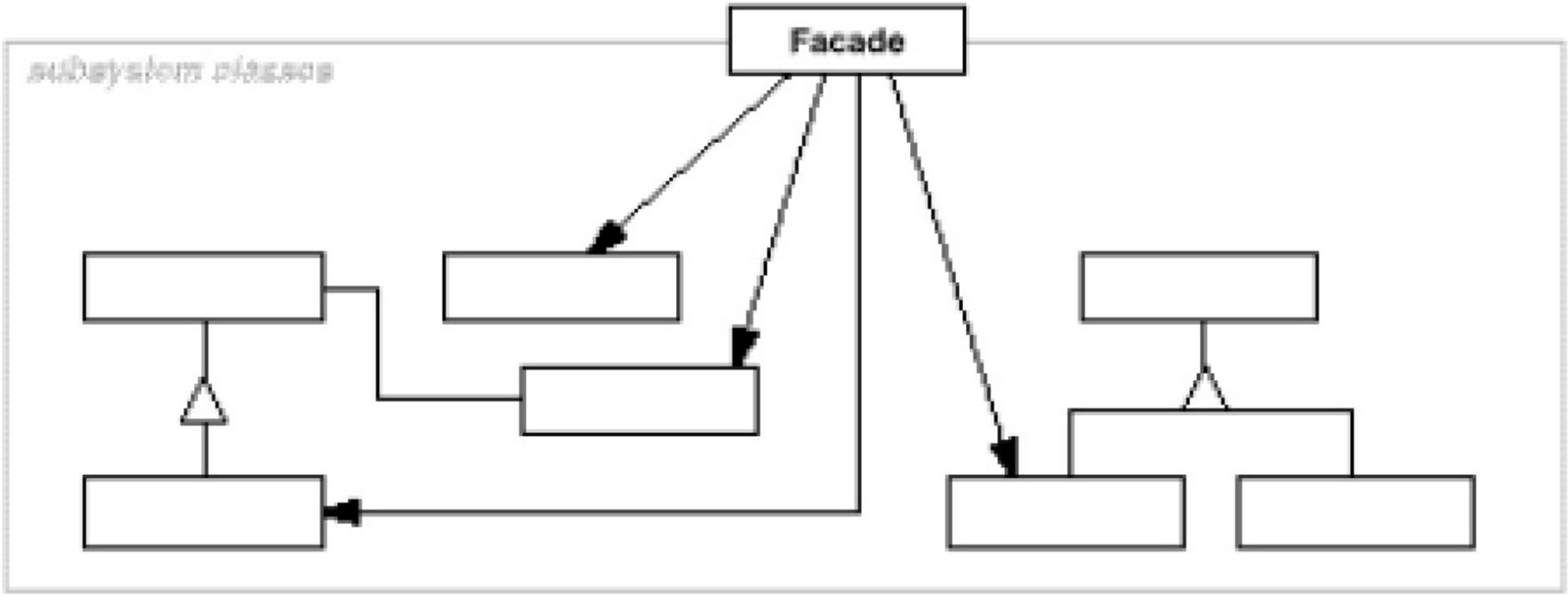


Patron « Adapter » - diagramme de séquence

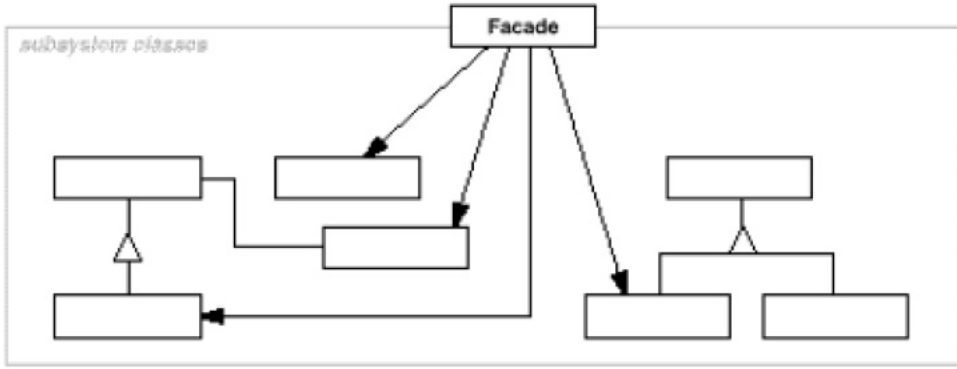


Facade


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
```

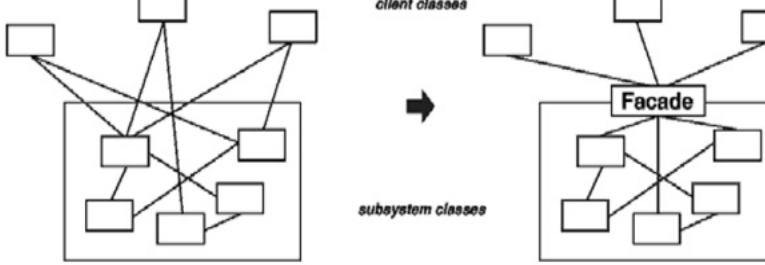
This class allows the Faces API to be unaware of the nature of its containing application environment. In particular, this class allows JavaServer Faces based appications to run in either a Servlet or a Portlet environment.

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.

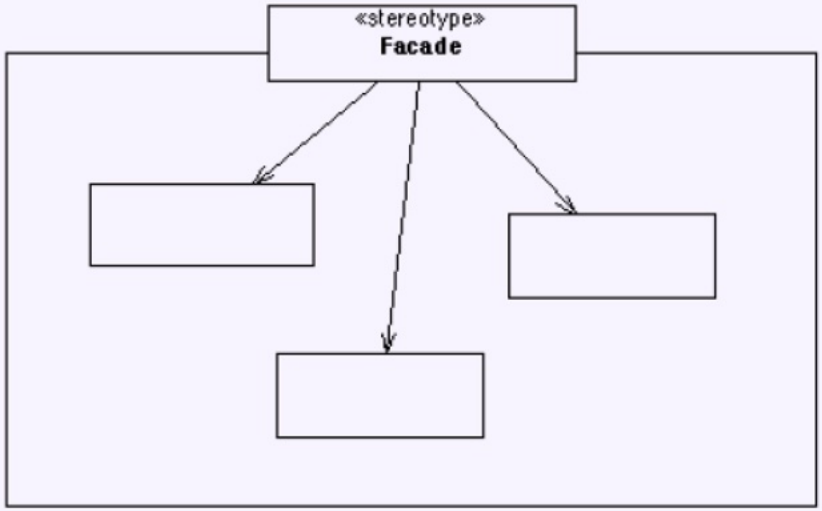
If a reference to an `ExternalContext` is obtained during application startup or shutdown time, any method documented as "valid to call this method during application startup or shutdown" must be supported during application startup or shutdown time. The result of calling a method during applicatic startup 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

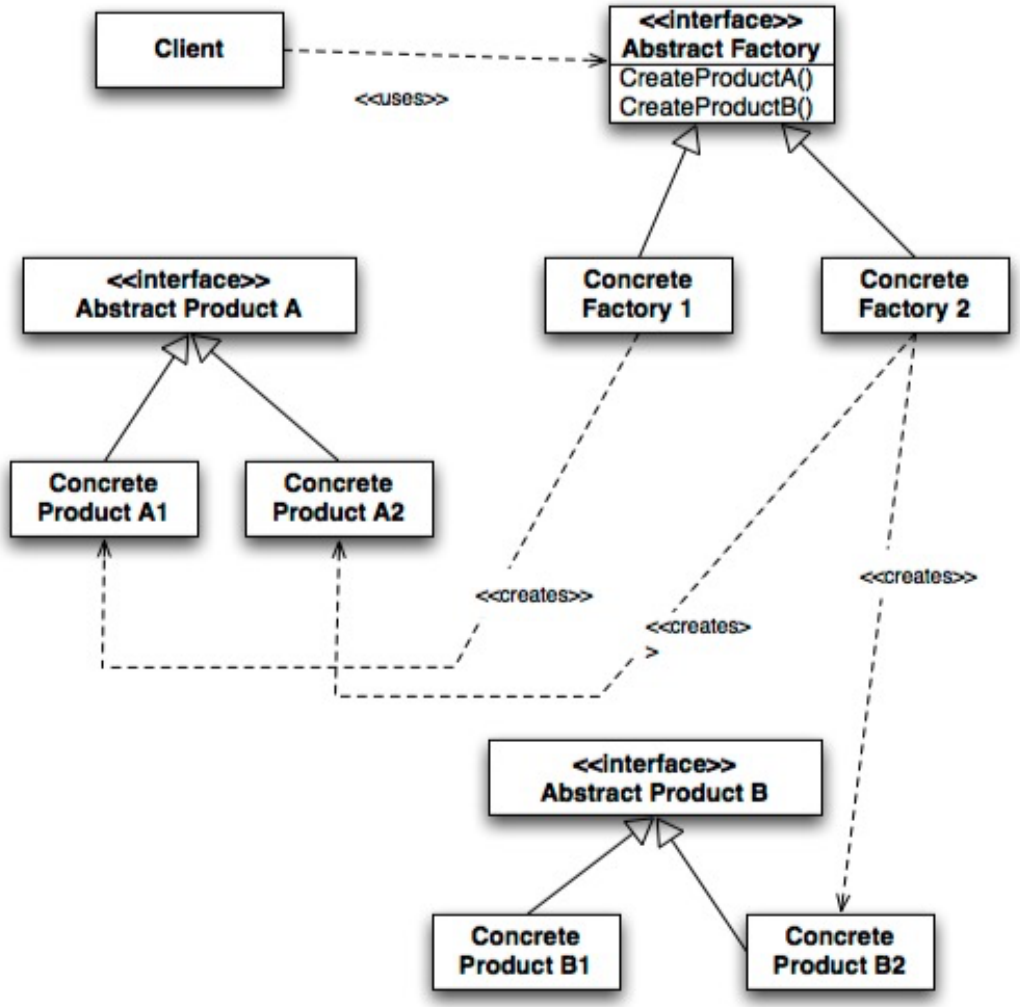
Patron « Abstract Factory » - rôles

- 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

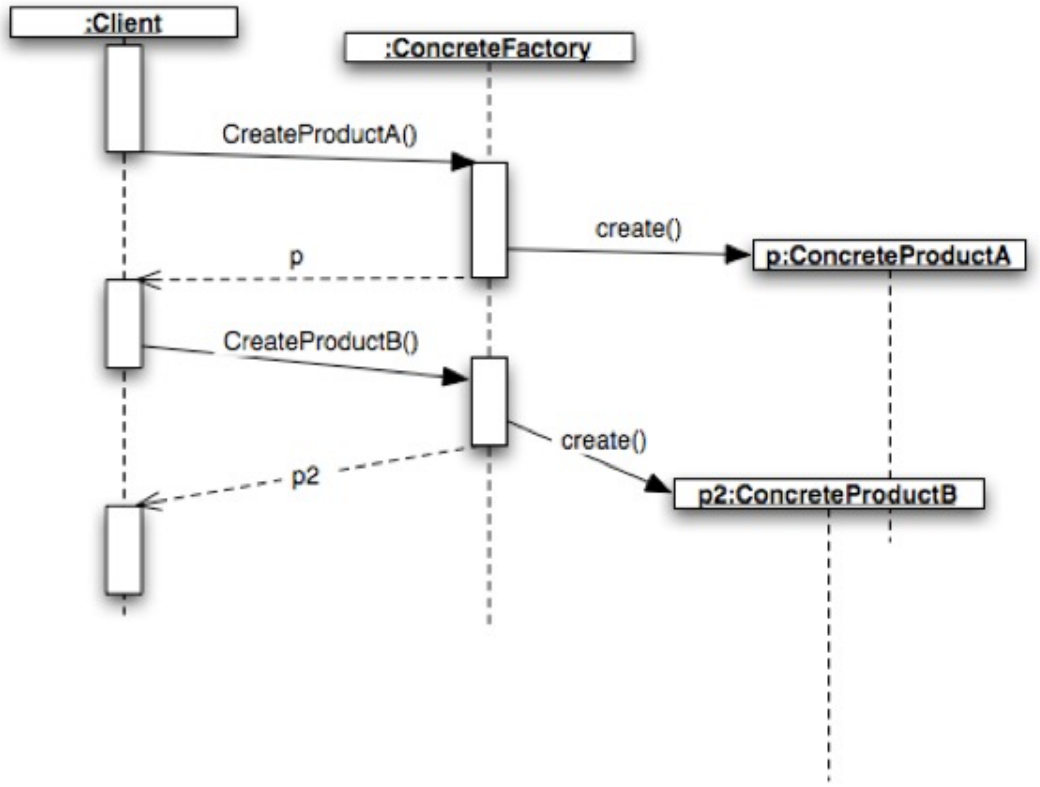
Patron « Abstract Factory » - rôles

- 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

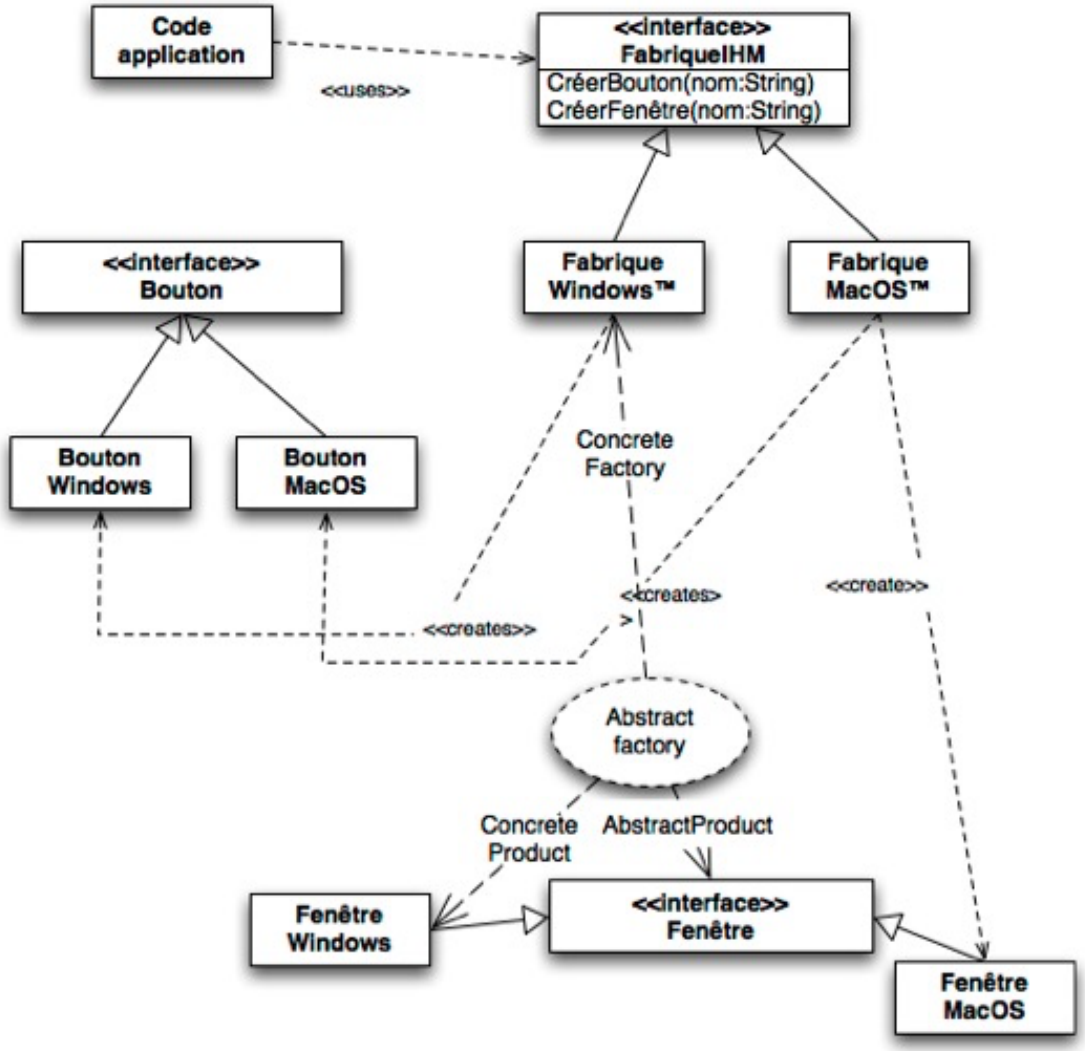
Patron « Abstract Factory » - Structure



Patron « Abstract Factory » - Collaboration



Patron « Abstract Factory » - Example



Adapter

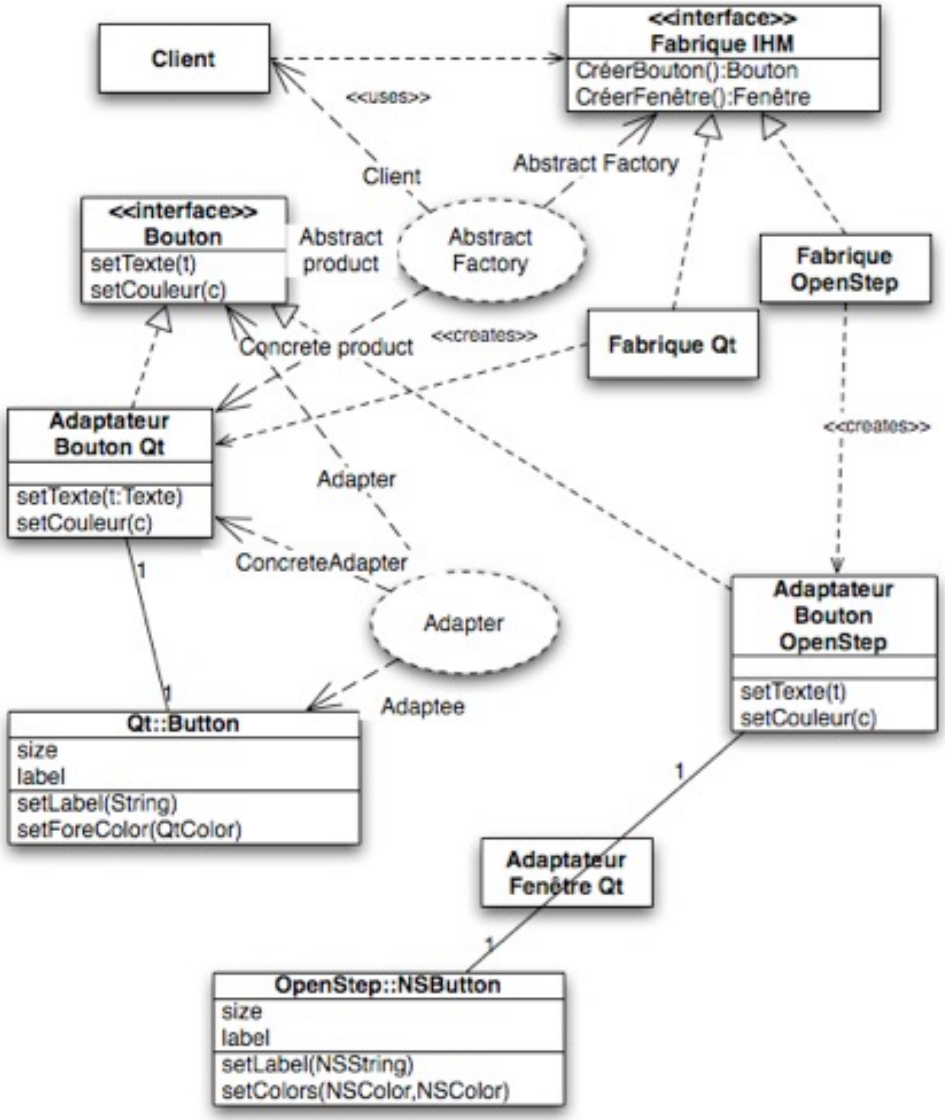
+

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 »

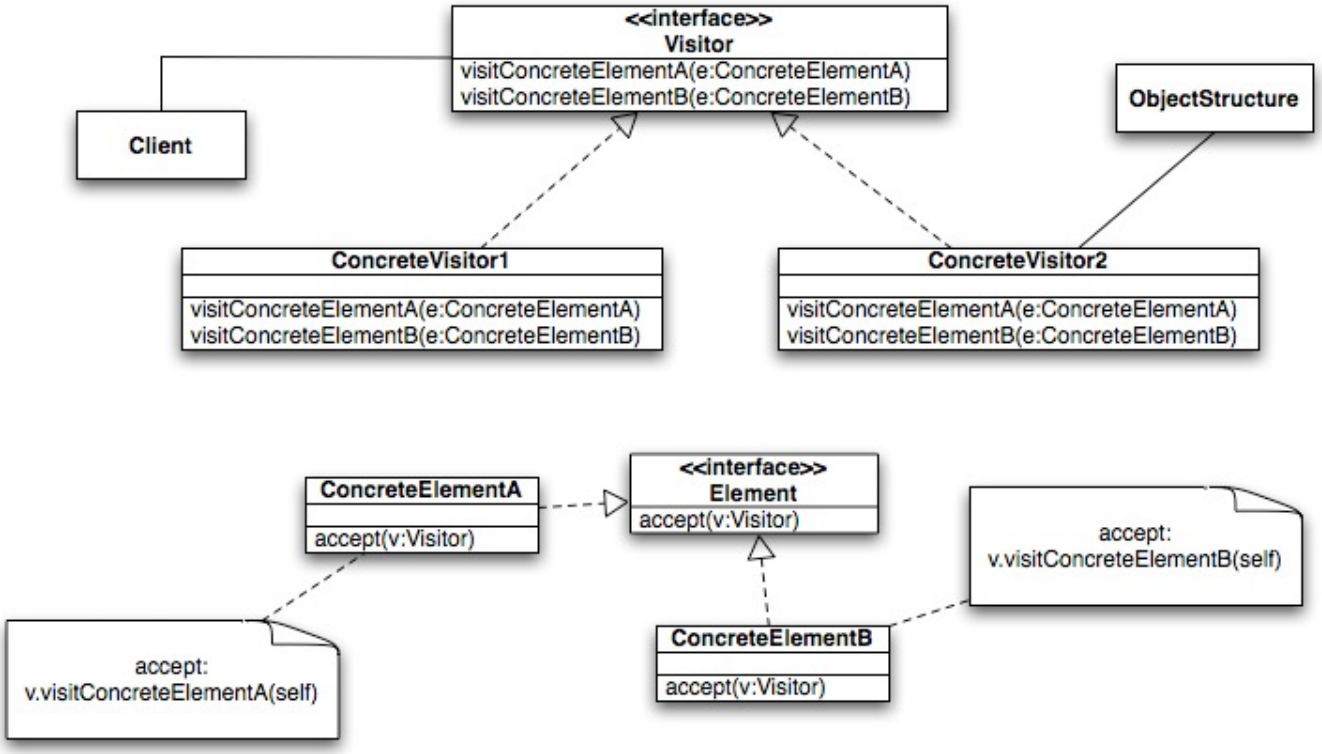


Visitor

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

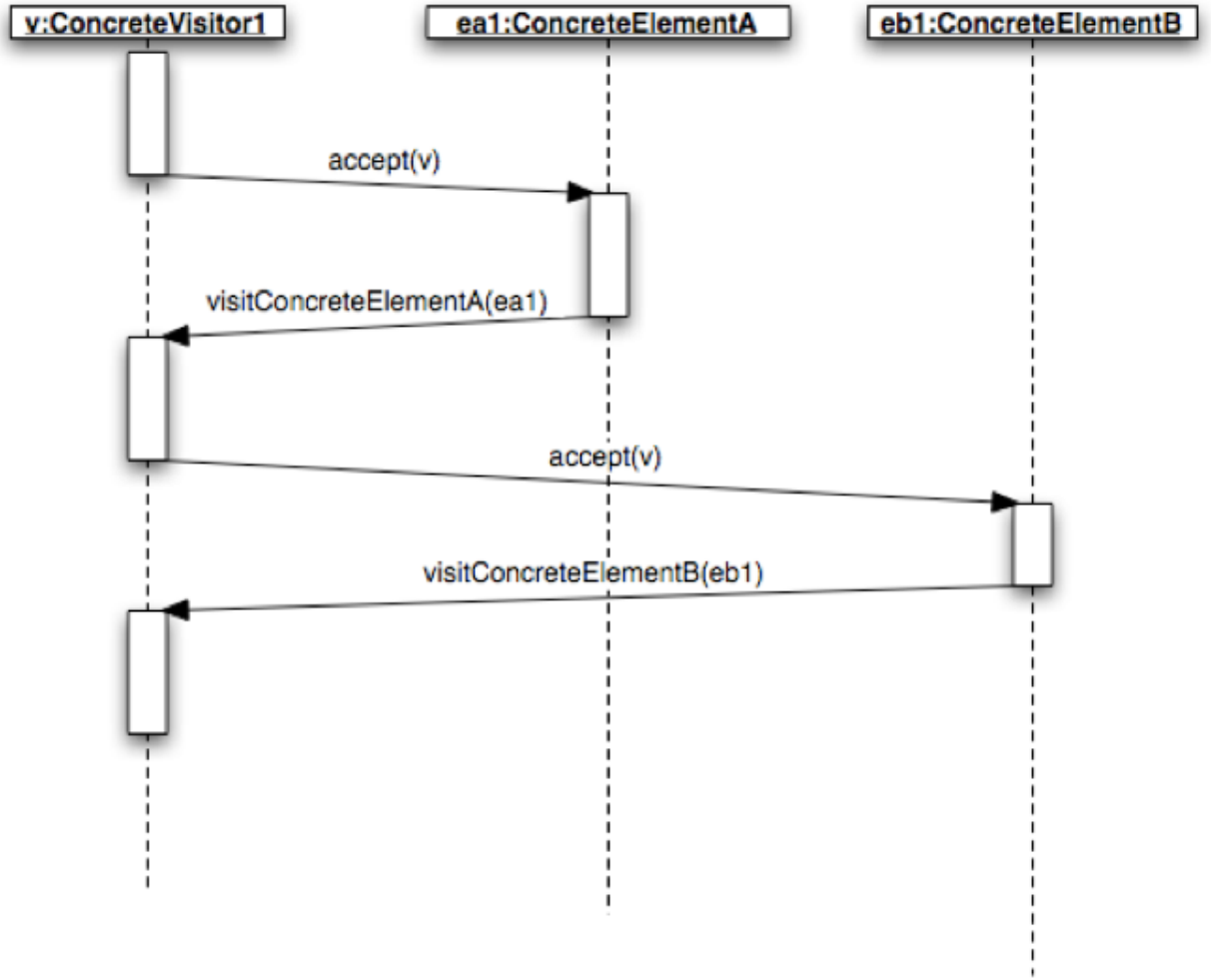
Patron « Visitor » - structure



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 » - 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

DIY

Patron « Visitor »: exercice

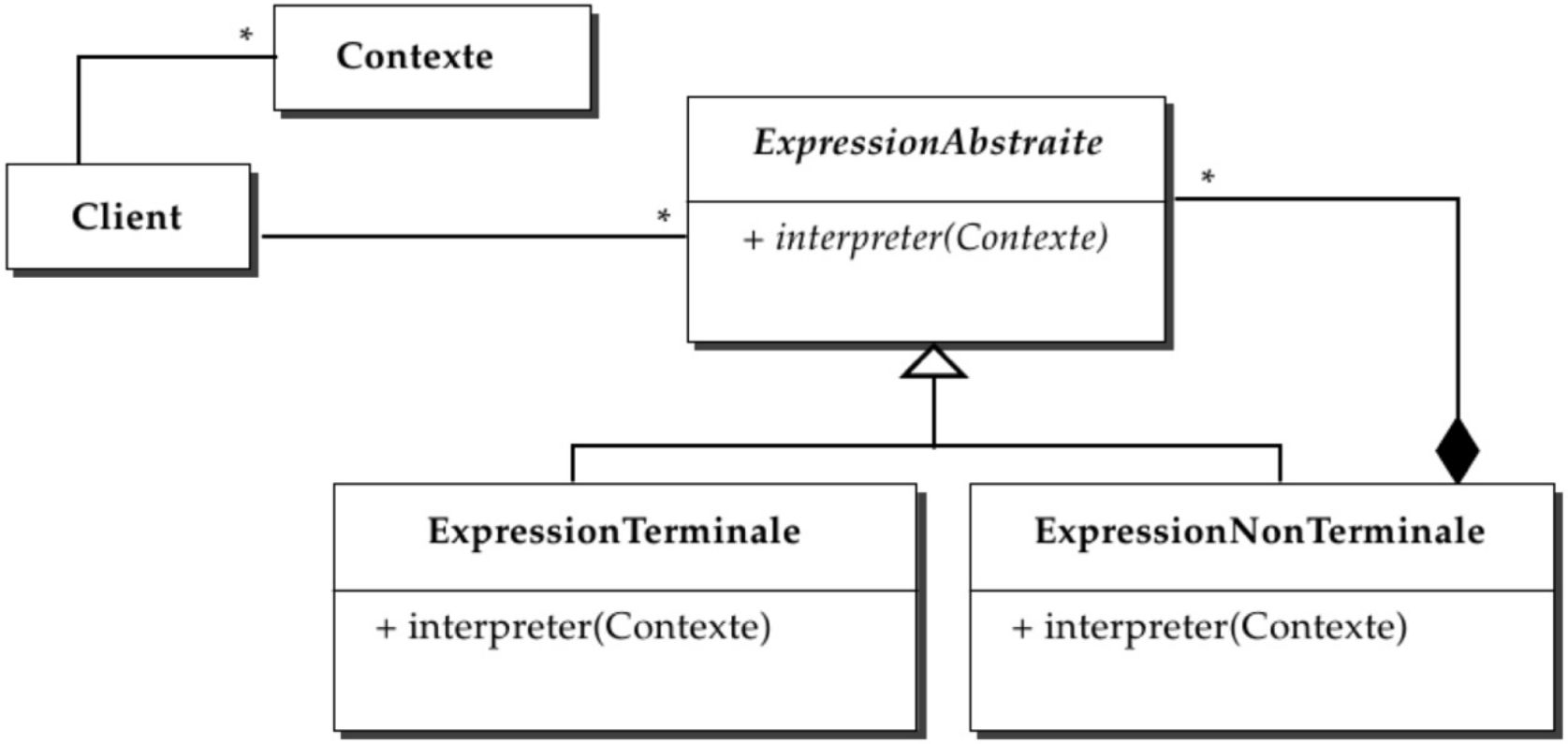
Dans cet exercice nous allons modéliser des expressions arithmétiques sous la forme d'un arbre binaire. Seules l'addition et la soustraction devront être considérées. Un *Arbre* possède un *Noeud racine* et un *nom*. Un *Noeud* est soit un *NoeudPlus*, soit un *NoeudMoins*, soit un *NoeudValeur*. *NoeudPlus* et *NoeudMoins* possède chacun un *noeud droit* et un *noeud gauche*. *NoeudValeur* possède une *valeur* (un entier). Vous modéliserez *Nœud* sous la forme d'une interface.

1. Créez le diagramme de classes de l'arbre.
2. Ajoutez le patron de conception Visiteur à l'arbre : ajoutez les méthodes accept(VisiteurArbre) à ce diagramme de classes et définissez l'interface VisiteurArbre et ses méthodes comme vu en cours.
3. Donnez le code Java de chacune des méthodes accept ajoutée.
4. Pourquoi les méthodes accept sont-elles nécessaires ?
5. Implémentez en Java un visiteur permettant d'afficher dans la console et en notation post-fixée la formule arithmétique que représente l'arbre.
6. Implémentez en Java un visiteur permettant de calculer la formule arithmétique que représente l'arbre. Un indice : le visiteur possédera une pile stockant les valeurs visitées. Vous utilisez alors le patron Visiteur pour faire un Interpréteur de manière propre.

Interpreter

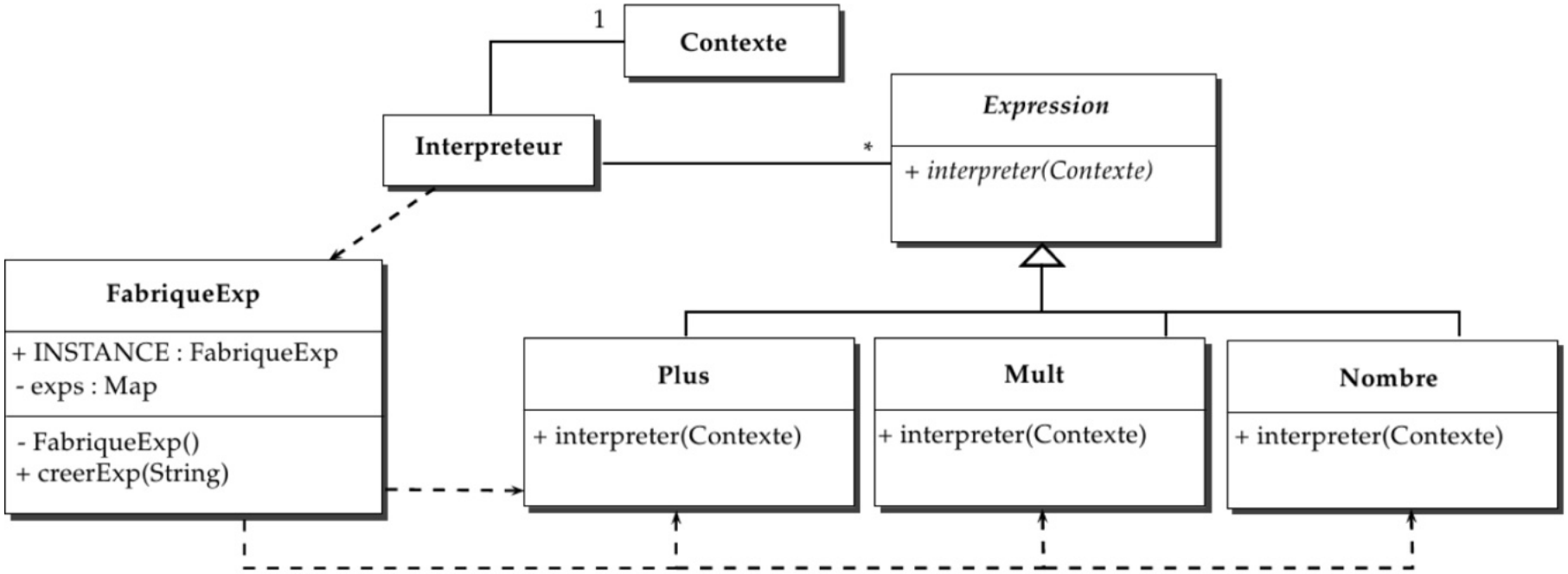
Patron de Conception
Interpréteur / Interpreter *(comportement)*

- But de l'Interpréteur :
  - Étant donné un langage, définir une représentation pour sa grammaire ainsi qu'un interpréteur pour interpréter des séquences du langage



Patron de Conception
Interpréteur / Interpreter *(comportement)*

- Exemple : arithmétique (notation polonaise inversée)



Patron de Conception
Interpréteur / Interpreter *(comportement)*

- Exemple : arithmétique (notation polonaise inversée)

```
class Contexte extends Stack<Double> {

  public Contexte() {
    super();
  }

  public double getFinalValue() {
    if(size()==1)
      return peek();
    return Double.NaN;
  }
}
```

```
interface Expression {
  void interprete(Contexte ctxt);
}
```

```
class Mult implements Expression {
  public void interprete(Contexte ctxt) {
    ctxt.push(ctxt.pop()*ctxt.pop());
  }
}
```

```
class Nombre implements Expression {
  protected double value;

  public Nombre(double val) {
    super();
    value = val;
  }

  public void interprete(Contexte ctxt) {
    ctxt.push(value);
  }
}
```

```
class Plus implements Expression {
  public void interprete(Contexte ctxt) {
    ctxt.push(ctxt.pop()+ctxt.pop());
  }
}
```

A Case of Visitor versus Interpreter Pattern

Mark Hills[1,2], Paul Klint[1,2], Tijs van der Storm[1], and Jurgen Vinju[1,2]

[1] Centrum Wiskunde & Informatica, Amsterdam, The Netherlands
[2] INRIA Lille Nord Europe, France

2.1 Creating and Processing Abstract Syntax Trees

Rascal has many AST classes (about 140 abstract classes and 400 concrete classes). To facilitate language evolution the code for these classes, along with the Rascal parser, is generated from the Rascal grammar. The AST code generator also creates a Visitor interface (`IASTVisitor`), containing methods for all the node types in the hierarchy, and a default visitor that returns null for every node type (`NullASTVisitor`). This class prevents us from having to implement a visit method for all AST node types, especially useful when certain algorithms focus on a small subset of nodes. Naturally, each AST node implements the `accept(IASTVisitor<T> visitor)` method by calling the appropriate visit method. For example, `Statement.If` contains:

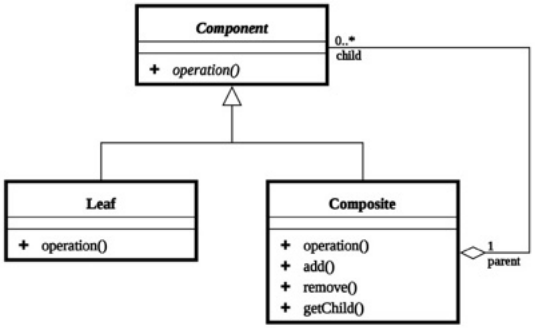


Fig. 2. The Composite Pattern[3]

```
public <T> accept(IASTVisitor<T> v) {
    return v.visitStatementIf(this);
}
```

The desire to generate this code played a significant role in initially deciding to use the Visitor pattern. We wanted to avoid having to manually edit generated code. Using the Visitor pattern, all functionality that operates on the AST nodes can be separated from the generated code. When the Rascal grammar changes, the AST hierarchy is regenerated. Many implementations of `IASTVisitor` will contain Java compiler errors and warnings because the signature of visit methods will have changed. This is very helpful for locating the code that needs to be changed due to a language change. Most of the visitor classes actually extend `NullASTVisitor`

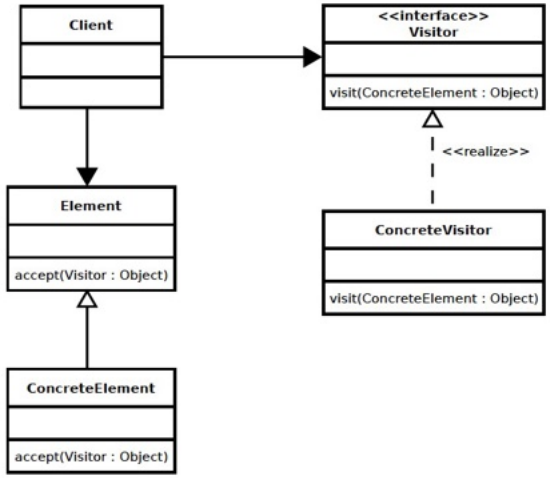


Fig. 3. The Visitor Pattern[4]

2.2 A Comparison with the Interpreter Pattern

Considering that our design already employs the Composite pattern, the difference in design complexity between the Visitor and Interpreter patterns is striking (Figure 4). The Composite pattern contains all the elements for the Interpreter pattern (abstract classes that are instantiated by concrete ones)—only an `interpret` method needs to be added to all relevant classes. So rather than having to add new concepts, such as a `Visitor` interface, the `accept` method and `NullASTVisitor`, the Interpreter pattern builds on the existing infrastructure of Composite and reuses it. Also, by adding more `interpret` methods (varying either the name or the static type) it is possible to reuse the Interpreter design pattern again and again without having to add additional classes. However, as a consequence, understanding each algorithm as a whole is now complicated by the fact that the methods implementing it are scattered over different AST classes. Additionally, there is the risk that methods contributing to different algorithms get tangled because a single AST class may have to manage the combined state required for all implemented algorithms. The experiments discussed in Section 4 help make this tradeoff between separation of concerns and complexity more concrete.

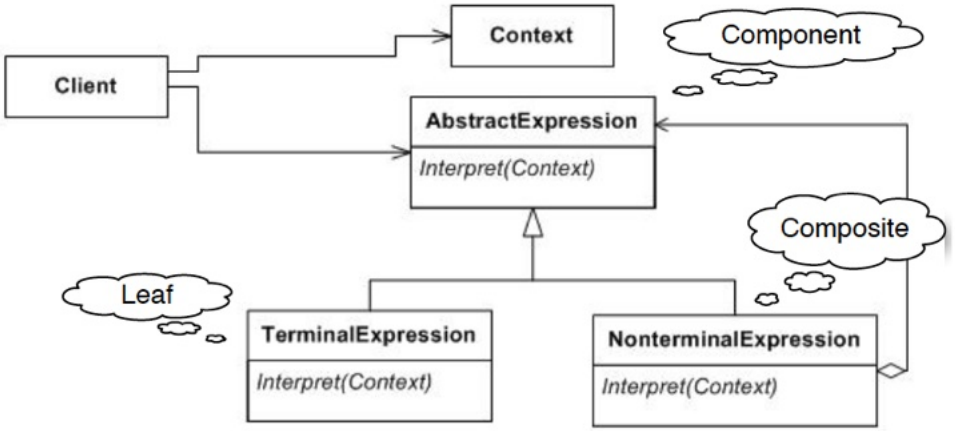


Fig. 4. The Interpreter Pattern with references to Composite (Figure 2).[7]

Memento

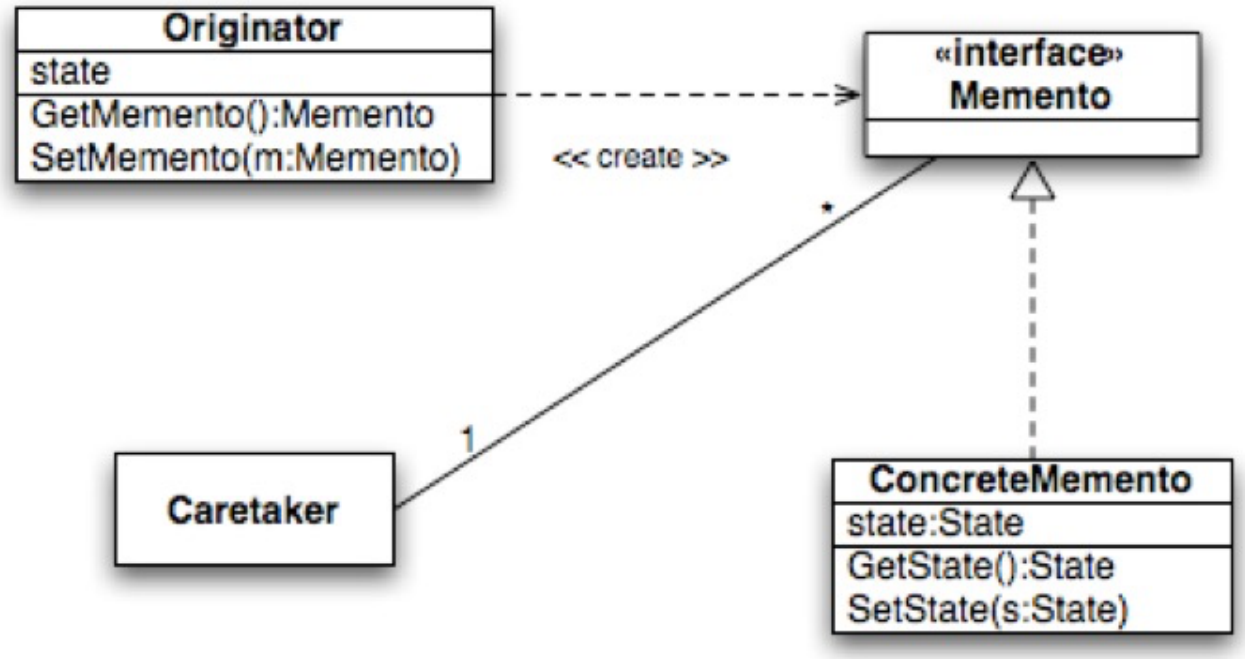
Patron « Memento »

- 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

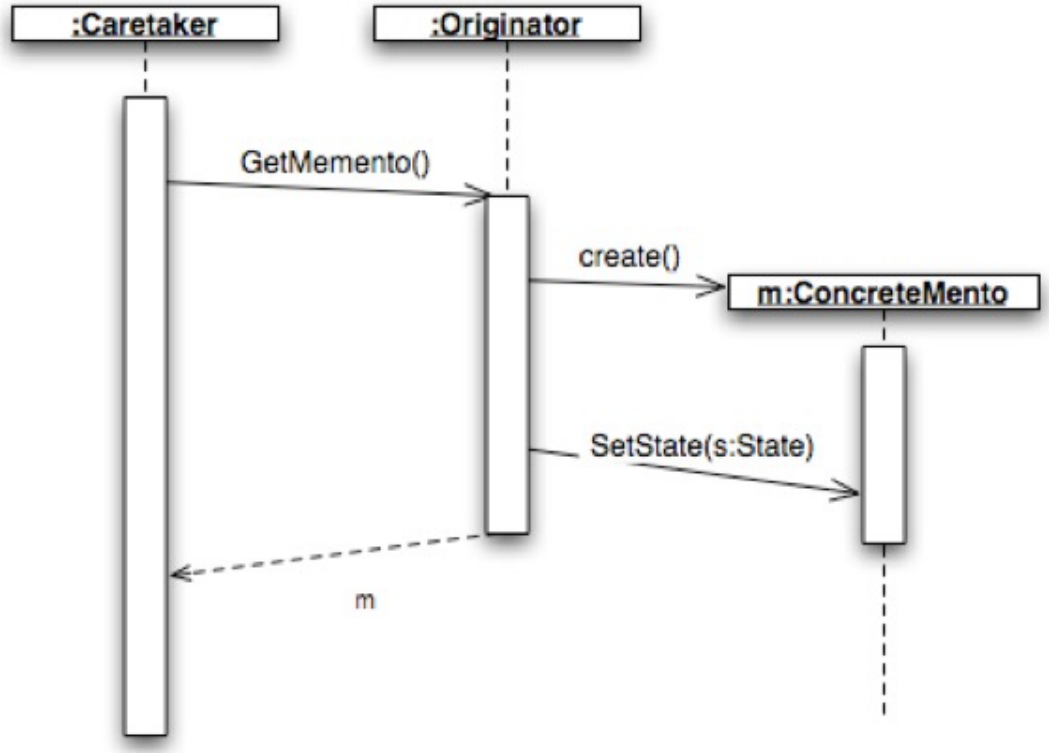
Patron « Memento » - rôles

- 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

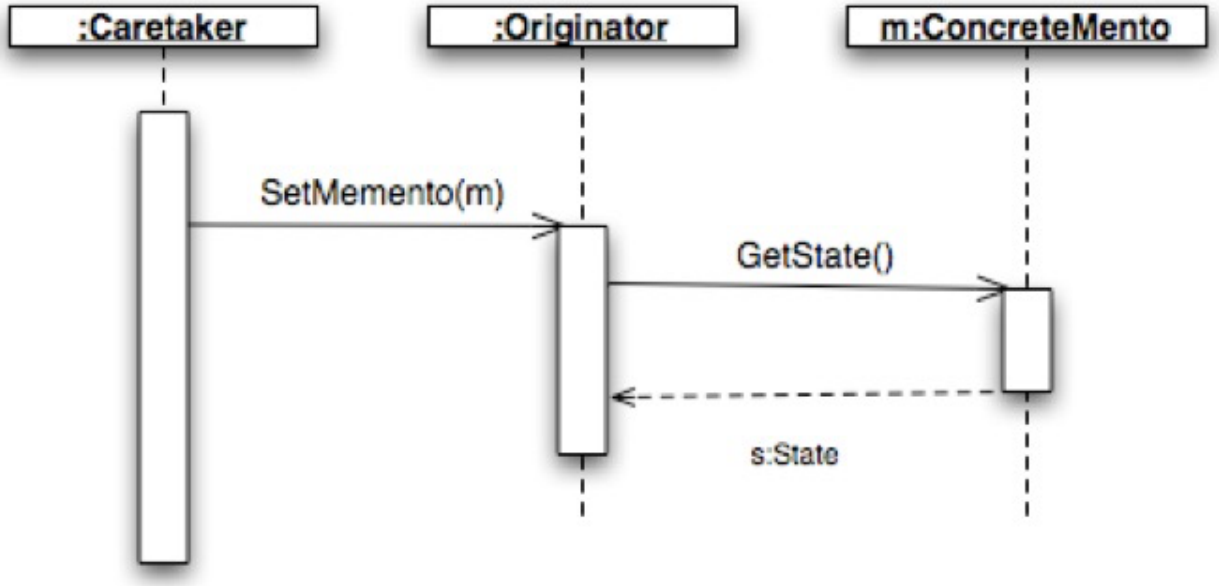
Patron « Memento » - structure



Patron « Memento » - sauvegarde



Patron « Memento » - restauration



Flyweight
(poids mouche)

Poids-Mouche / Flyweight *(structurel)*

Problème:

Instanciation d’un (très très) grand nombre de petits objets

- Trop gourmand au niveau de la mémoire

Exemples :

Du texte contenant un grand nombre de caractères

Un jeu massivement multi-joueurs (les objets graphiques)

L'ADN

Patron de Conception
Poids-Mouche / Flyweight *(structurel)*

But :

Partager efficacement un grand nombre de petits objets

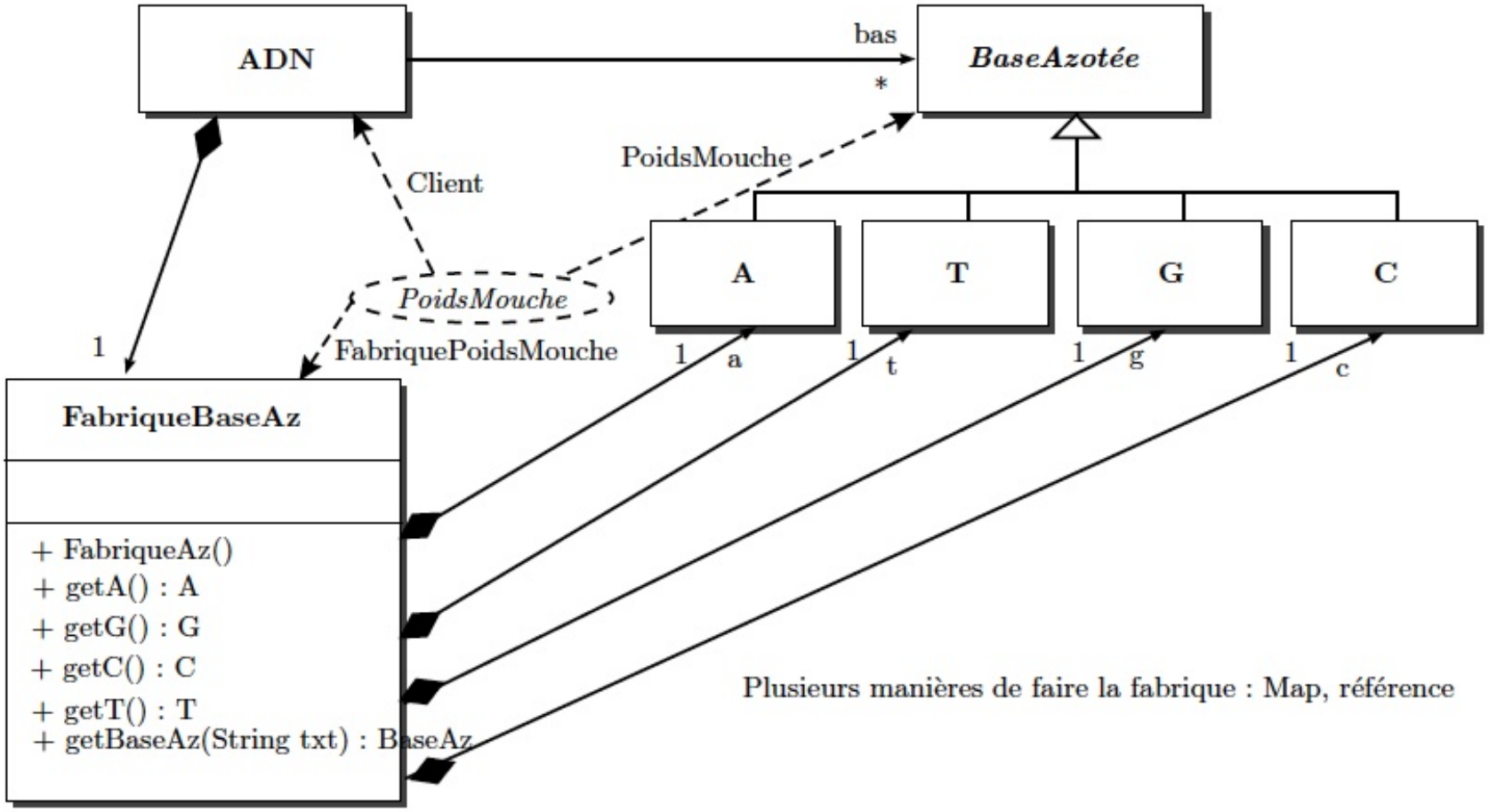
Fonctionnement :

Certaines données sont extraites de l'objet
pour en minimiser le nombre d'instanciation

Exemple : l'ADN

Seuls 4 bases azotées créées (G, A, T, C)
Leur position est stockée en dehors (dans l'ADN)

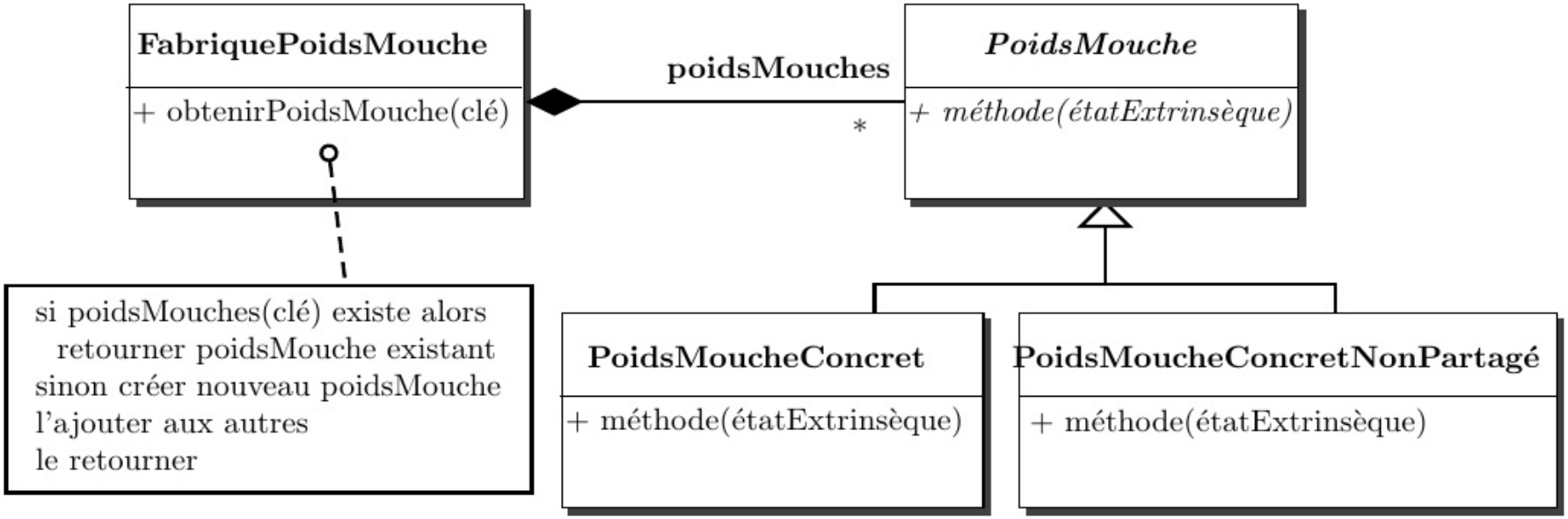
=> Sinon explosion du nombre d'instances de bases azotées



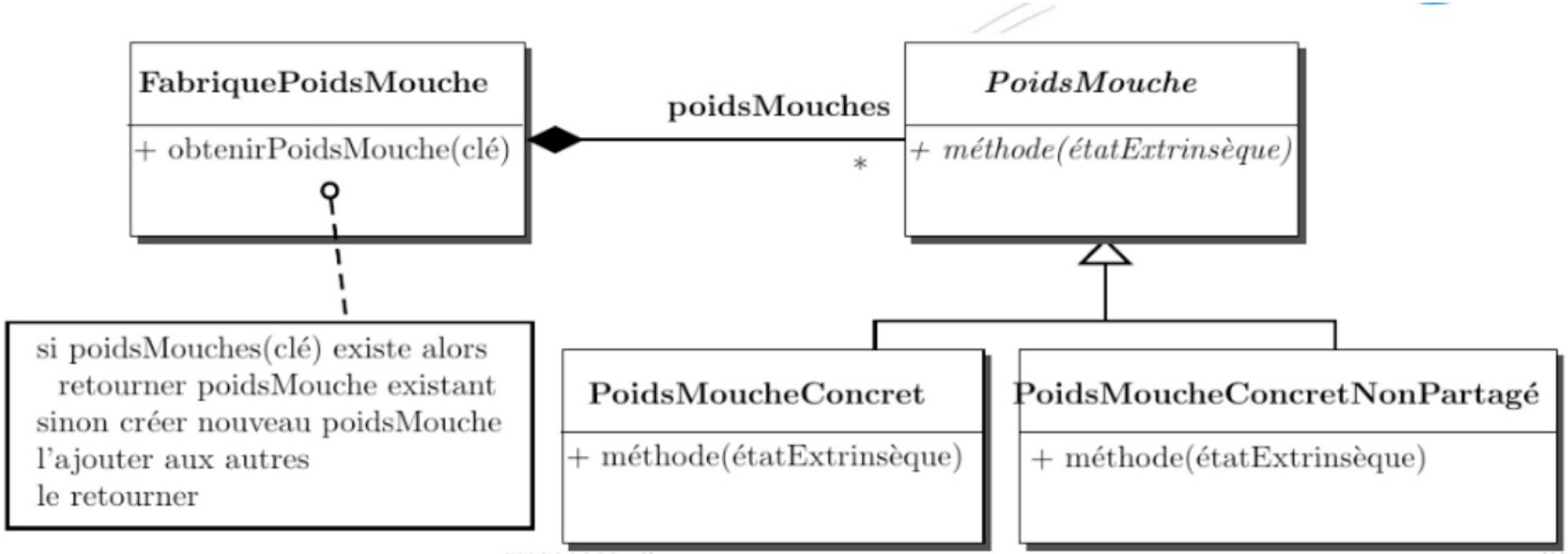
Plusieurs manières de faire la fabrique : Map, référence

```
public class FabriqueBaseAz {
        private A a;
        private C c;
        private G g;
        private T t;

    public FabriqueBaseAz() {
                super();// Ou alors lazy instanciation
                a = new A(); c = new C();
        g = new G(); t = new T();
        }

        public A getA() {
                return a;
        }
        public C getC() {
                return c;
        }
        public G getG() {
                return g;
        }
        public T getT() {
                return t;
        }
}
```

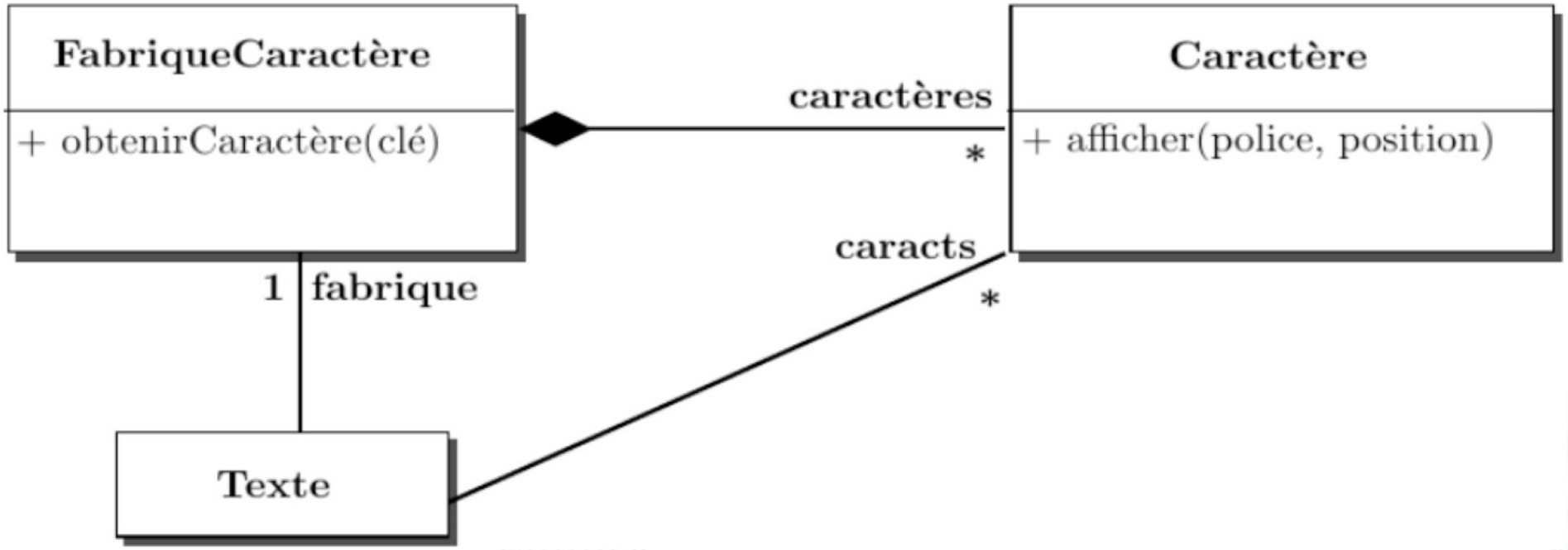
Poids-Mouche / Flyweight *(structurel)*



FabriquePoidsMouche : créer des objets uniquement si nécessaire (sinon retourne l'objet déjà existant)
L'utilisation du patron dans le code client
passe par la fabrique pour obtenir des instances



- **PoidsMoucheConcret** ne contient pas certaines données
  - ➔ Elles lui seront passées en paramètres (*étatExtrinsèque*)
- **PoidsMoucheConcretNonPartagé**: cas spécial de poids-mouche qui n'externalise pas ses données



- Exemple : un texte se compose de caractères
  - Externalisation de la position/police des caractères
  - 1 seul caractère pour 1 valeur (a, b, c, *etc.*)

```java
public final class FabriqueCaractere {
  public static FabriqueCaractere INSTANCE = new FabriqueCaractere();

  private Map<Integer, Caractere> mapCaracteres;

  private FabriqueCaractere() {
    mapCaracteres = new IdentityHashMap<Integer, Caractere>();
  }


  public Caractere ObtenirCaractere(char valeur) {
    Caractere car = mapCaracteres.get(valeur);

    if(car==null) {
      car = new Caractere(valeur);
      mapCaracteres.put((int)valeur, car);
    }
    return car;
  }
}
```

```java
public class Caractere {
  protected char valeur;
  //..

  public Caractere(char valeur) {
    this.valeur = valeur;
  }

  public void afficher(Graphics2D g, Point position) {
    //...
  }
}
```

```java
public class Texte {
  protected List<Caractere> caracts;

  public Texte() {
    caracts = new ArrayList<Caractere>();
  }

  public void afficher(Graphics2D g) {
    //...
  }

  public void ajouterCaractere(char valeur) {
    caracts.add(FabriqueCaractere.INSTANCE.ObtenirCaractere(valeur));
```

Builder

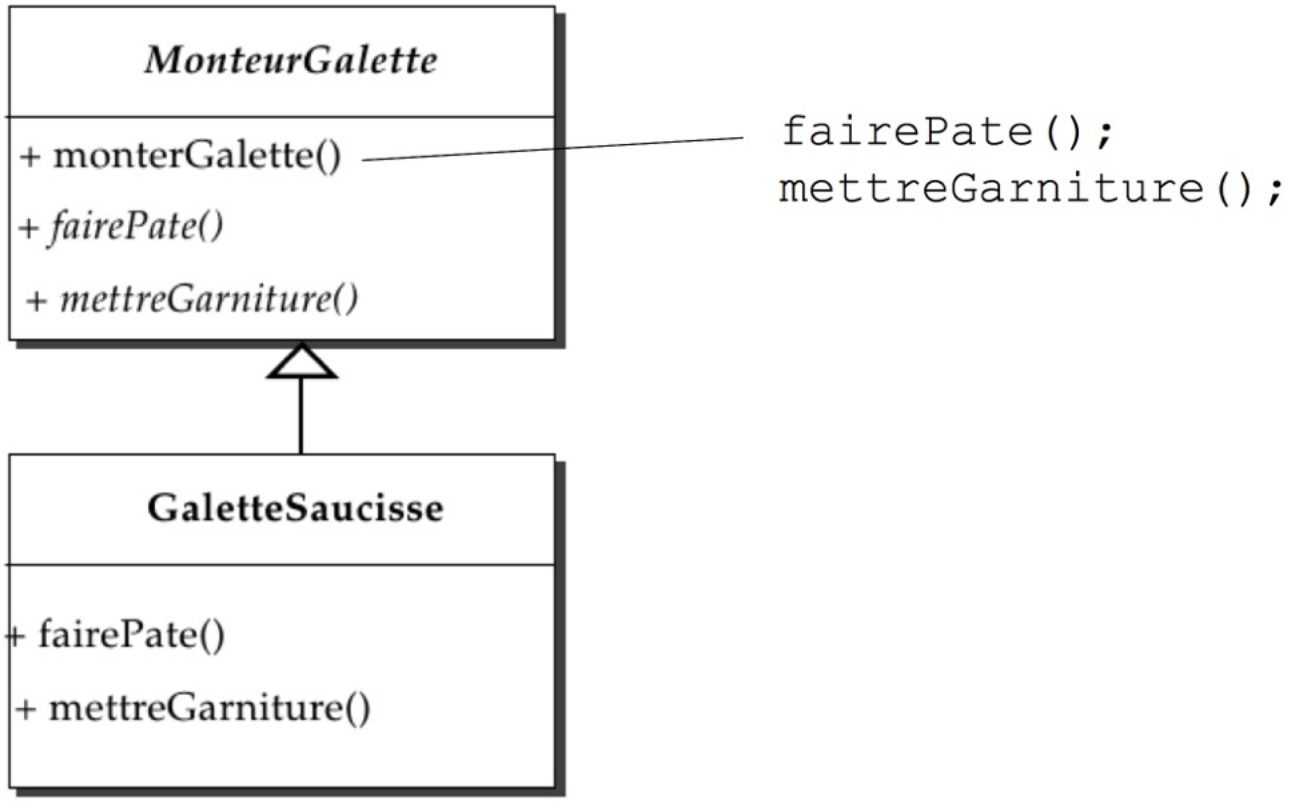
Patron de Conception
Monteur / Builder *(création)*

- Problème :
  - La construction de certains objets peut être complexe
  - → Mettre cette construction dans la classe concernée l'alourdirait

- Exemples :
  - Le montage d'un ordinateur
    - → Plusieurs configurations possibles
  - Création de grilles de Sudoku
    - → Plusieurs niveaux de difficulté possibles

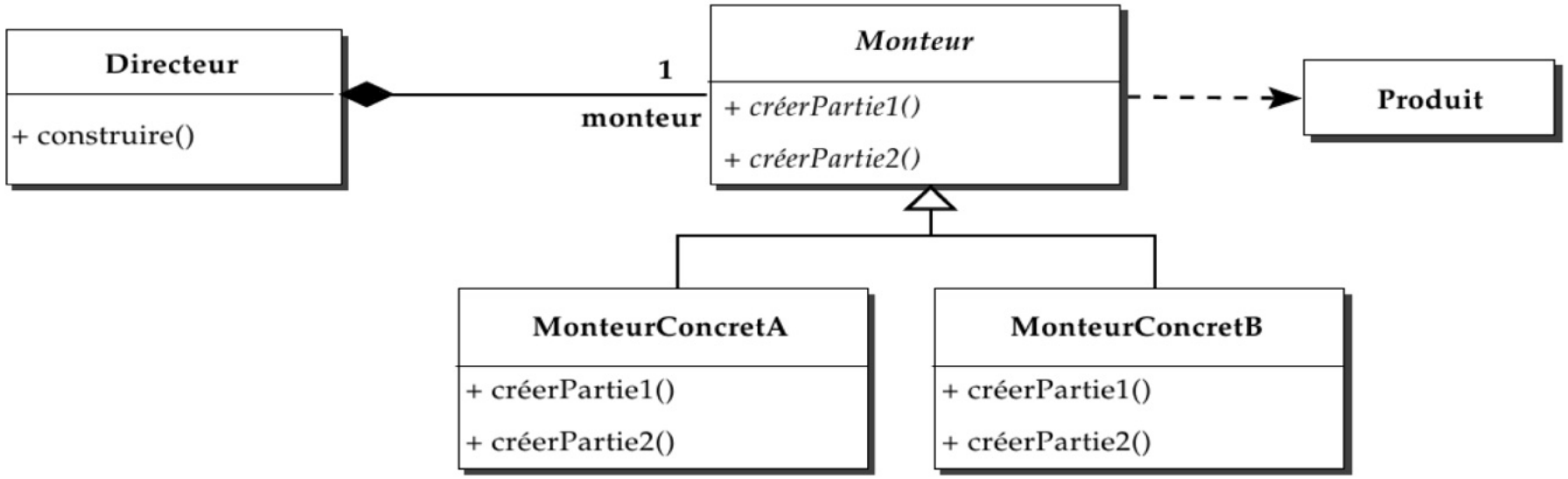
Patron de Conception
Patron de méthode /
Template method *(comportement)*

- Exemple : la création de galettes



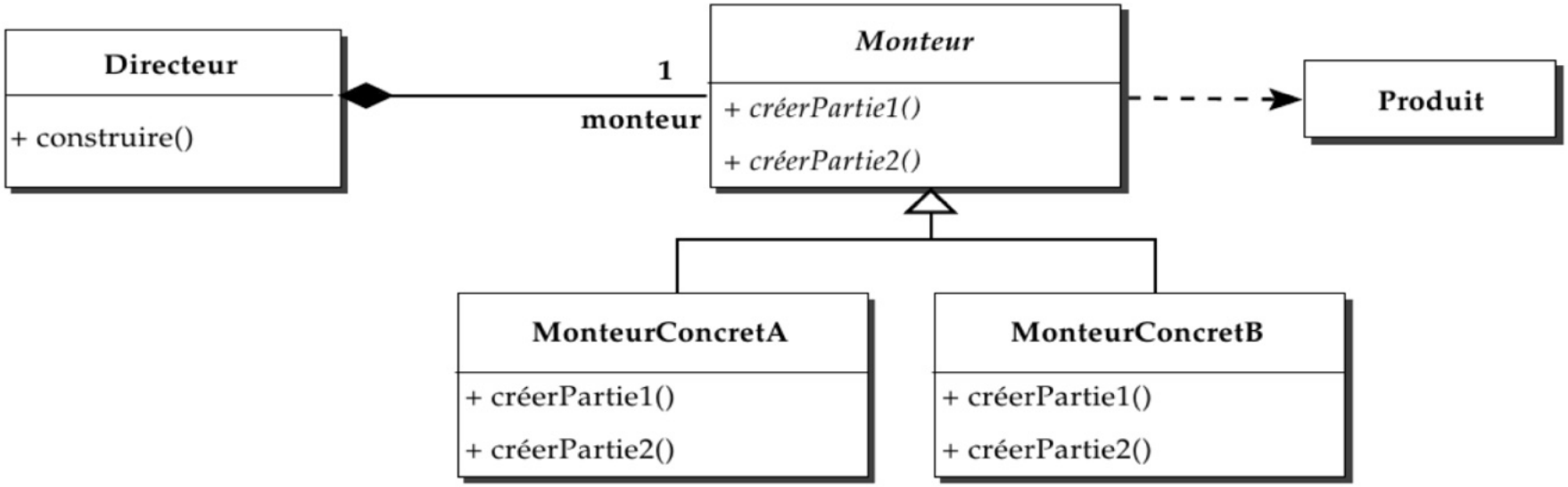
Patron de Conception
Monteur / Builder *(création)*

- But :
  - Séparer la création d'objets complexes de leur représentation
  - Le même processus de création peut servir à créer différents objets complexes



Utilise très souvent le patron *Patron de méthode*

Patron de Conception
Monteur / Builder *(création)*



- **Directeur** : définit le processus de création
- **Monteur** : interface pour l'ajout de parties dans le **Produit**
- **MonteurConcret** : différentes implémentations de création

Patron de Conception
Monteur / Builder *(création)*

Exemple d'un monteur dans JavaFX

Tiré de : http://blog.netopyr.com/2012/01/24/advantages-of-javafx-builders/

(JavaFX est la librairie graphique de Java, en remplacement de Java Swing)

Code Java pour créer et paramétrer des widgets de manière classique (sans *Monteur*) :

```
Text text1 = new Text(50, 50, "Hello World!");
text1.setFill(Color.WHITE);
text1.setFont(MY_DEFAULT_FONT);

Text text2 = new Text(50, 100, "Goodbye World!");
text2.setFill(Color.WHITE);
text2.setFont(MY_DEFAULT_FONT);

Text text3 = new Text(50, 150, "JavaFX is fun!");
text3.setFill(Color.WHITE);
text3.setFont(MY_DEFAULT_FONT);
```

- Cela fonctionne mais verbeux
- C'est un inconvénient de la programmation impérative

Patron de Conception
Monteur / Builder *(création)*

Exemple d'un monteur dans JavaFX

Code Java pour créer et paramétrer des widgets avec un **monteur** fournit par JavaFX :

```
Text text1 = TextBuilder.create().text("Hello World!").x(50).y(50)
                .fill(Color.WHITE).font(MY_DEFAULT_FONT).build();

Text text2 = TextBuilder.create().text("Goodbye World!").x(50).y(100)
                .fill(Color.WHITE).font(MY_DEFAULT_FONT).build();

Text text3 = TextBuilder.create().text("JavaFX is fun!").x(50).y(150)
                .fill(Color.WHITE).font(MY_DEFAULT_FONT).build();
```

- *create()* créer un monteur qui est ensuite paramétré
- *build()* construit ensuite l'instance de *Text*
- Cette manière de faire s'inspire de la **programmation fonctionnelle** :
  - enchainement des appels de fonctions sur un même objet
  - plus clair (en général), limite les effets de bord, facilite la parallélisation

Patron de Conception
Monteur / Builder *(création)*

Exemple d'un monteur dans JavaFX

Javadoc du monteur *TextBuilder* :

http://docs.oracle.com/javafx/2/api/javafx/scene/text/TextBuilder.html

Text build() : Make an instance of Text based on the properties set on this builder.

Static TextBuilder<?> create() : Creates a new instance of TextBuilder.

TextBuilder<?> font(Font x) : Set the value of the font property for the instance constructed by this builder.

```
public Text build() {
    Text x = new Text();
    applyTo(x); // Configure l'instance
    return x;
}
```

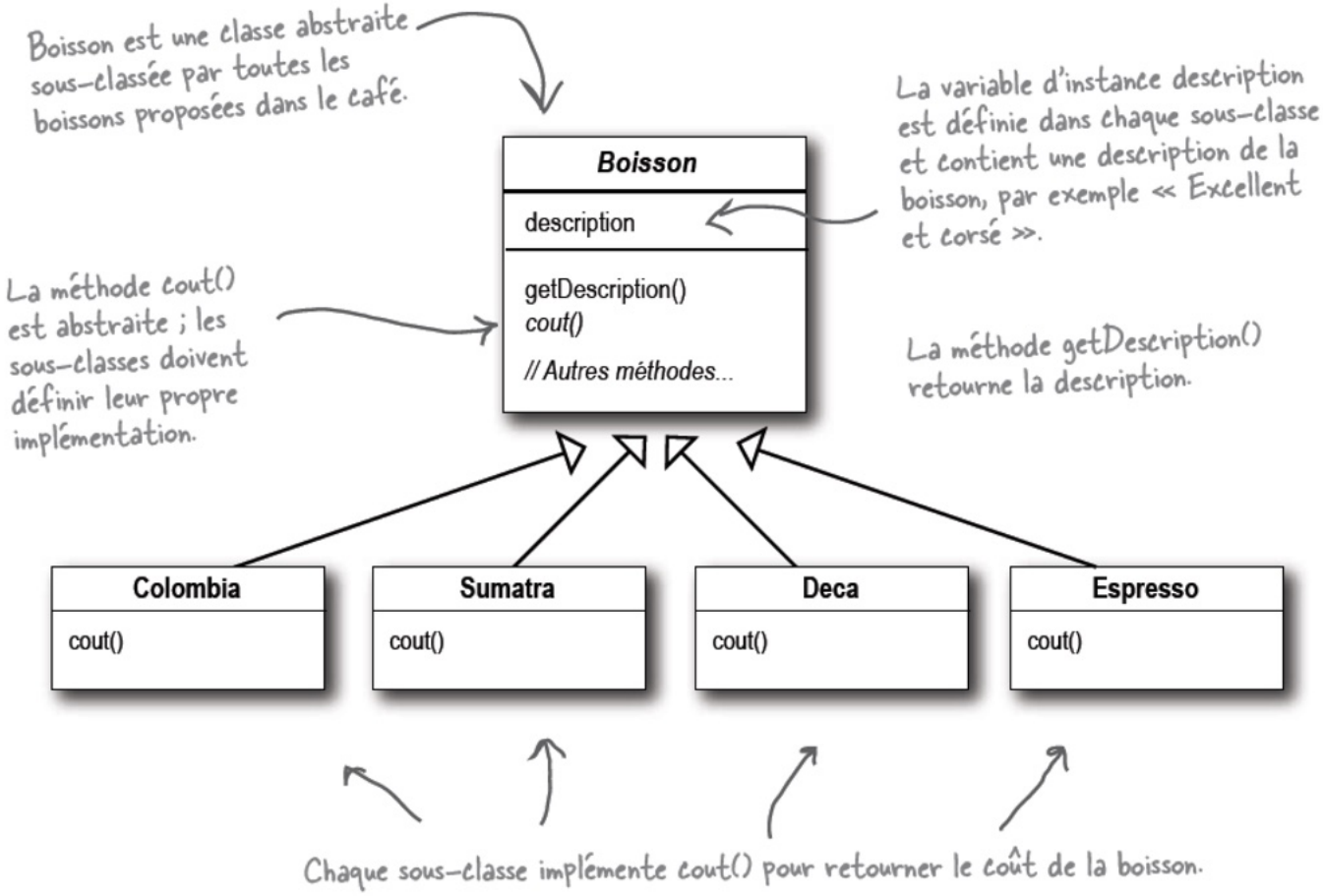
```
public static TextBuilder create() {
    return new TextBuilder();
}
```

```
public TextBuilder font(Font x) {
    font = x;
    return this;
}
```

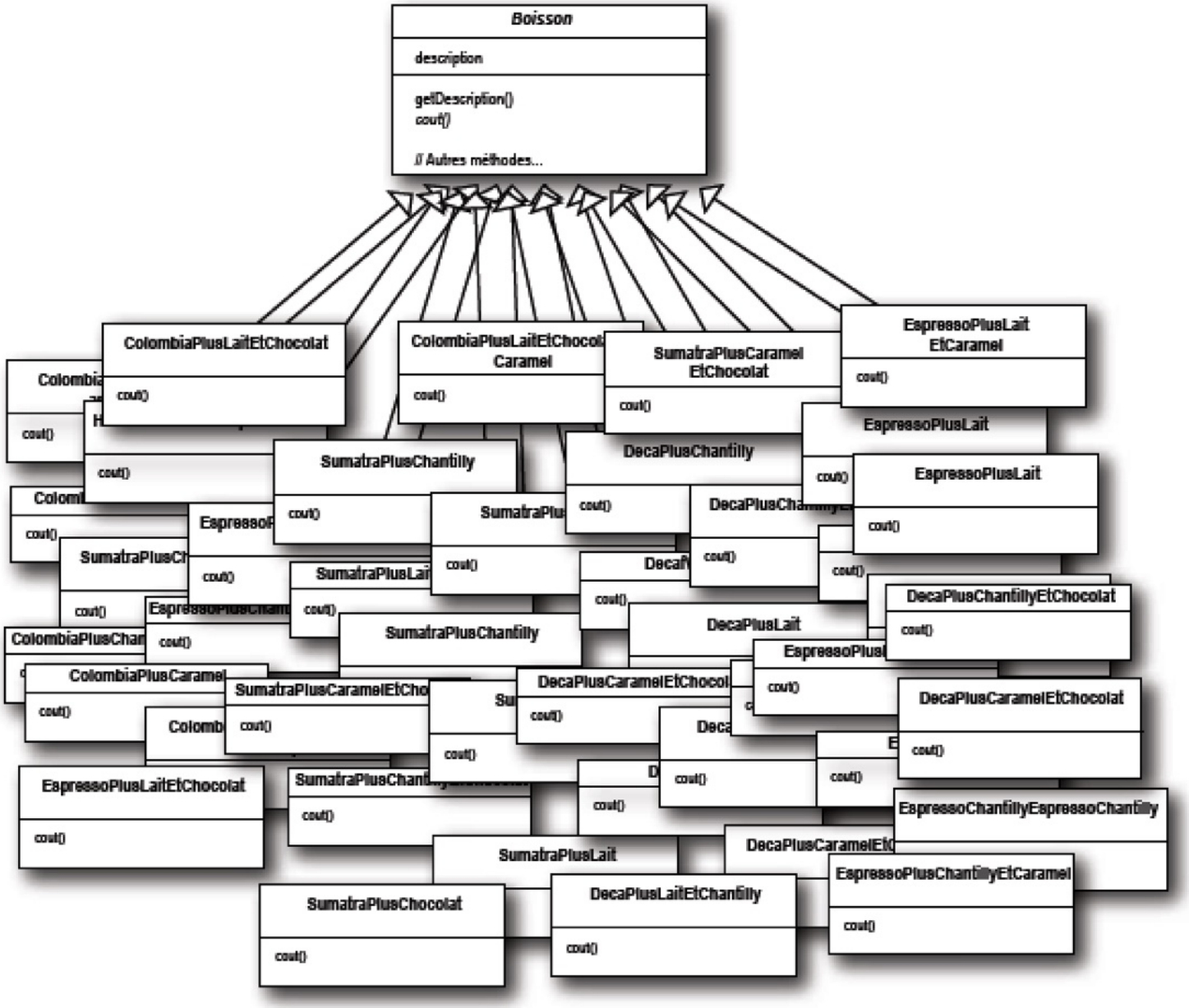
Les monteurs de JavaFX sont désormais «deprecated» (à ne plus utiliser) pour diverses raisons obscures : http://mail.openjdk.java.net/pipermail/openjfx-dev/2013-March/006725.html

Decorator

Un café?


Boisson est une classe abstraite sous-classée par toutes les boissons proposées dans le café.
Boisson
description
getDescription()
cout()
// Autres méthodes...
La variable d'instance description est définie dans chaque sous-classe et contient une description de la boisson, par exemple « Excellent et corsé ».
La méthode cout() est abstraite ; les sous-classes doivent définir leur propre implémentation.
La méthode getDescription() retourne la description.
Colombia
cout()
Sumatra
cout()
Deca
cout()
Espresso
cout()
Chaque sous-classe implémente cout() pour retourner le coût de la boisson.

#1 Héritage

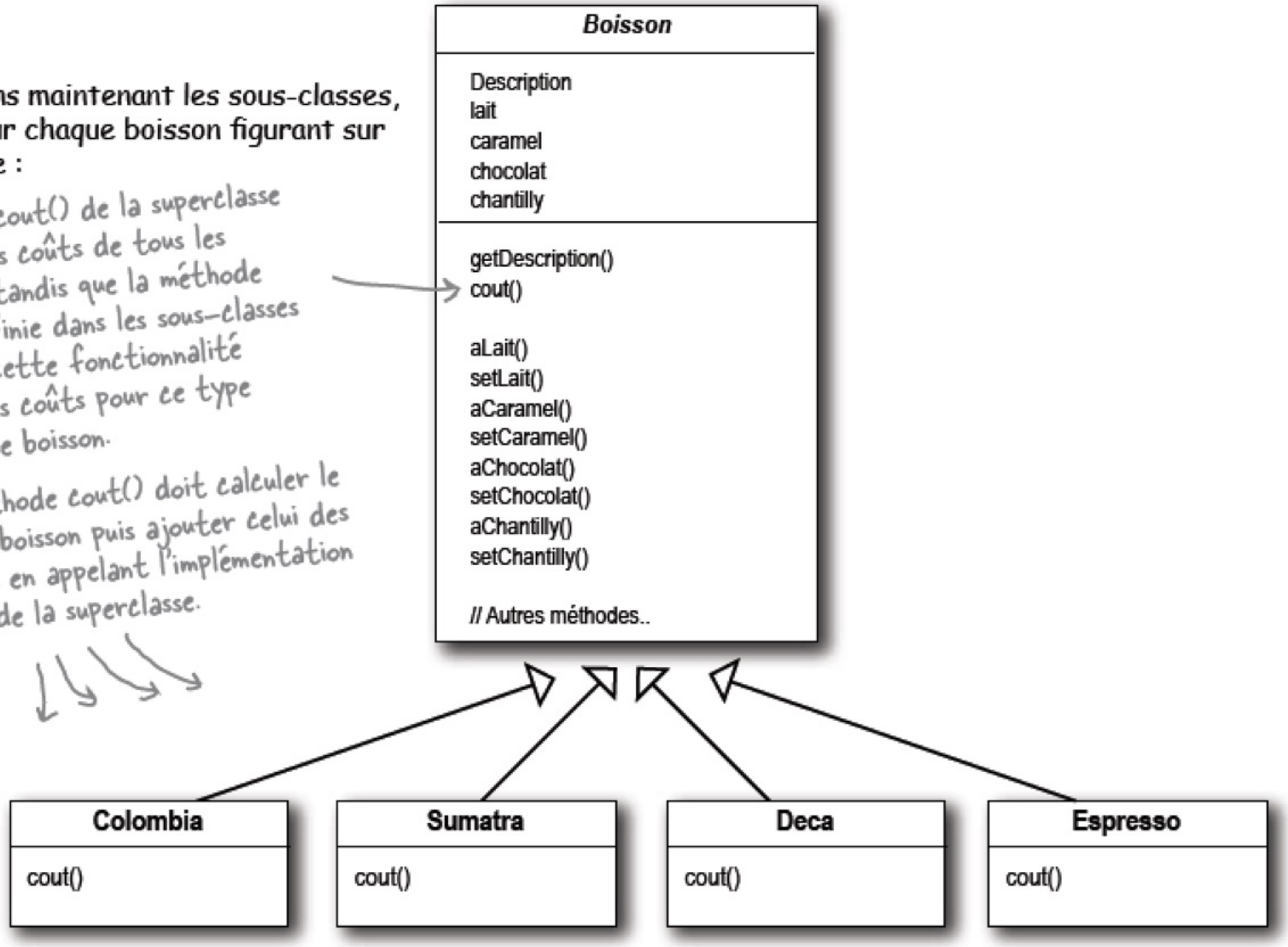

Boisson
description
getDescription()
cout()
// Autres méthodes...
ColombiaPlusLaitEtChocolat
cout()
SumatraPlusCaramel EtChocolat
cout()
EspressoPlusLait EtCaramel
cout()
EspressoPlusLait
cout()
SumatraPlusChantilly
cout()
DecaPlusChantilly
cout()
EspressoPlusLait
cout()
DecaPlusChantillyEtChocolat
cout()
DecaPlusLait
cout()
DecaPlusCaramelEtChocolat
cout()
EspressoPlusLaitEtChocolat
cout()
SumatraPlusLait
cout()
SumatraPlusChocolat
cout()
DecaPlusLaitEtChantilly
cout()
EspressoChantillyEspressoChantilly
EspressoPlusChantillyEtCaramel
cout()

#2 Une autre solution

Ajoutons maintenant les sous-classes, une pour chaque boisson figurant sur la carte :

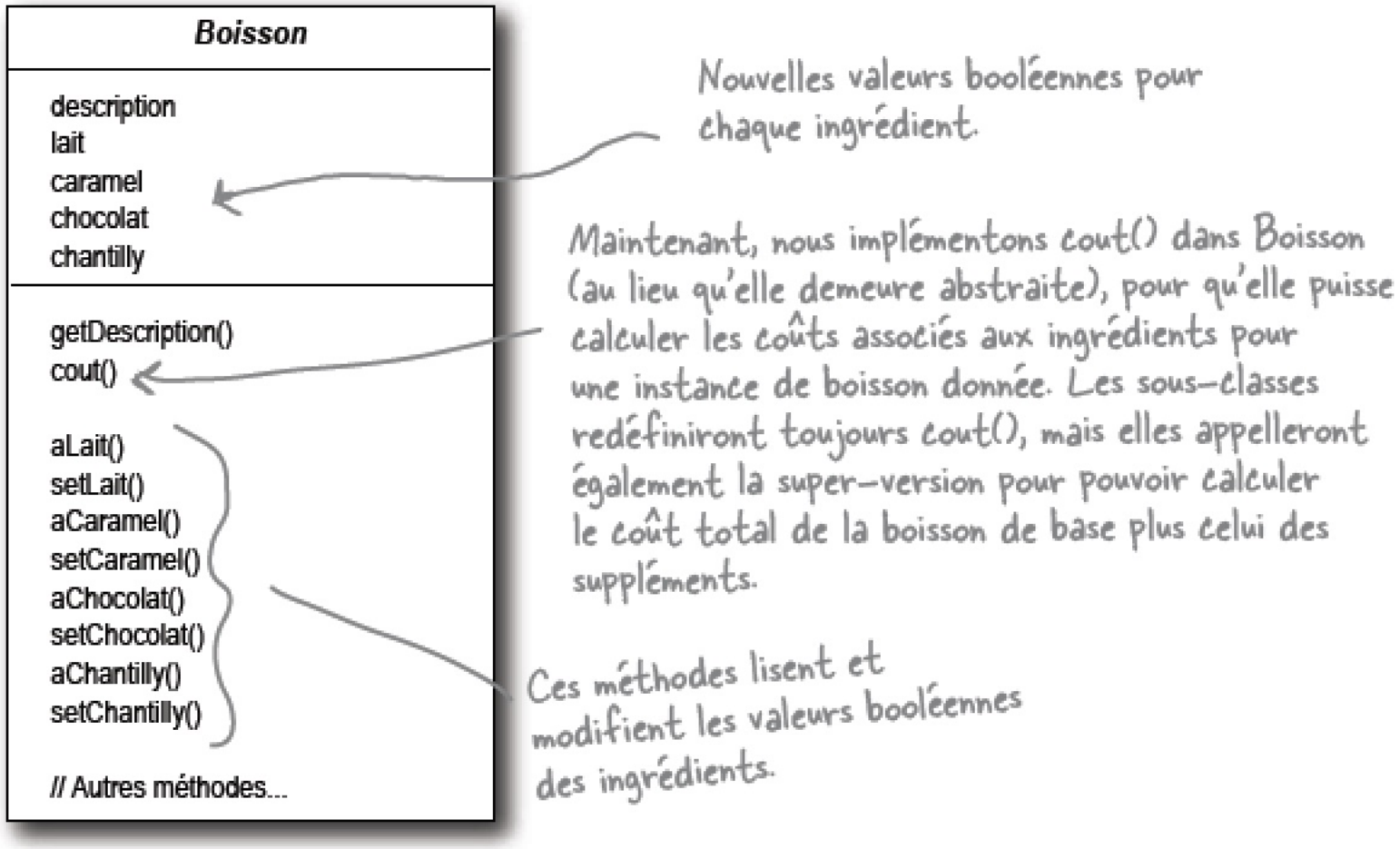
La méthode cout() de la superclasse va calculer les coûts de tous les ingrédients, tandis que la méthode cout() redéfinie dans les sous-classes va étendre cette fonctionnalité et inclure les coûts pour ce type spécifique de boisson.

Chaque méthode cout() doit calculer le coût de la boisson puis ajouter celui des ingrédients en appelant l'implémentation de cout() de la superclasse.

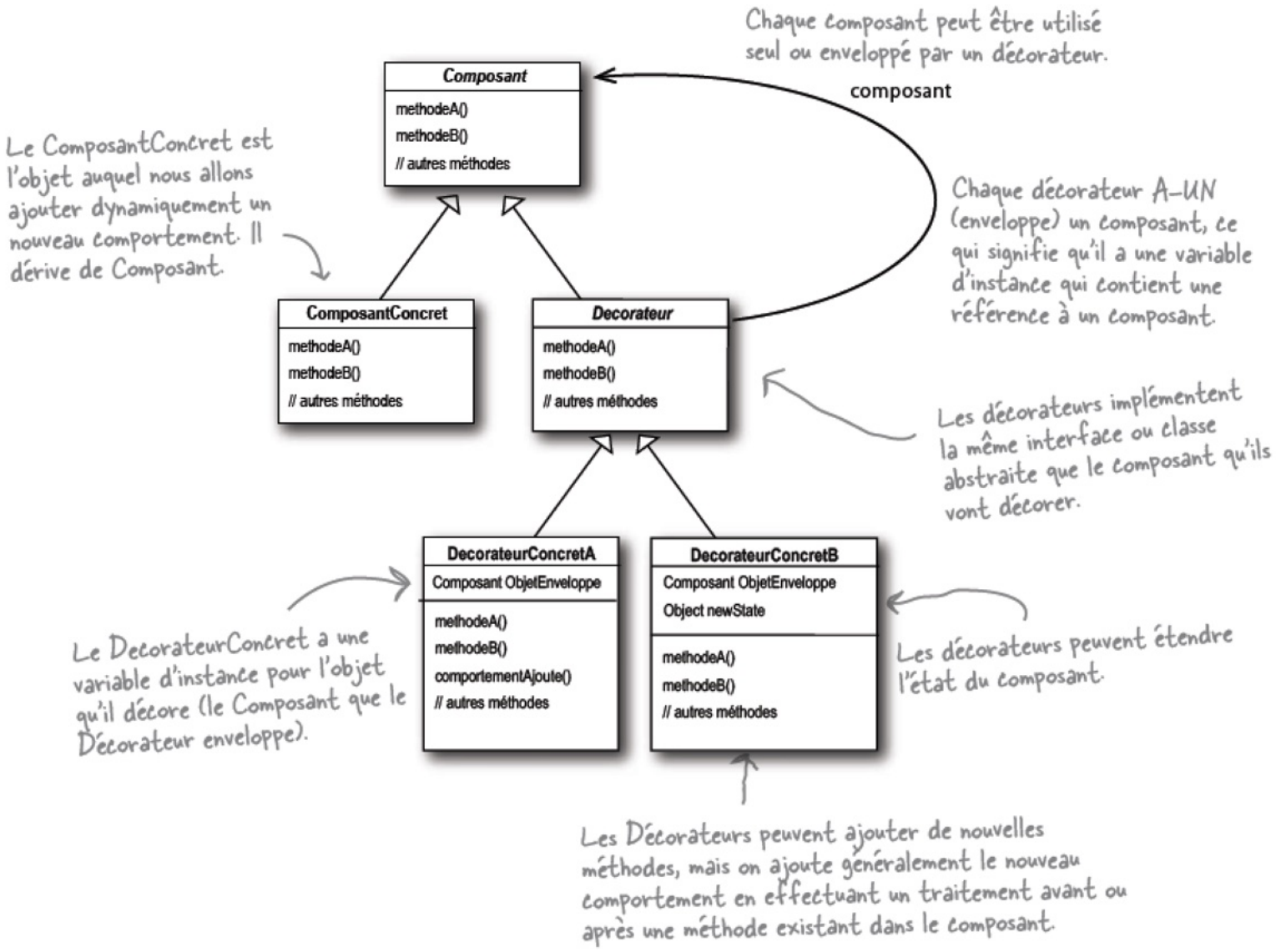


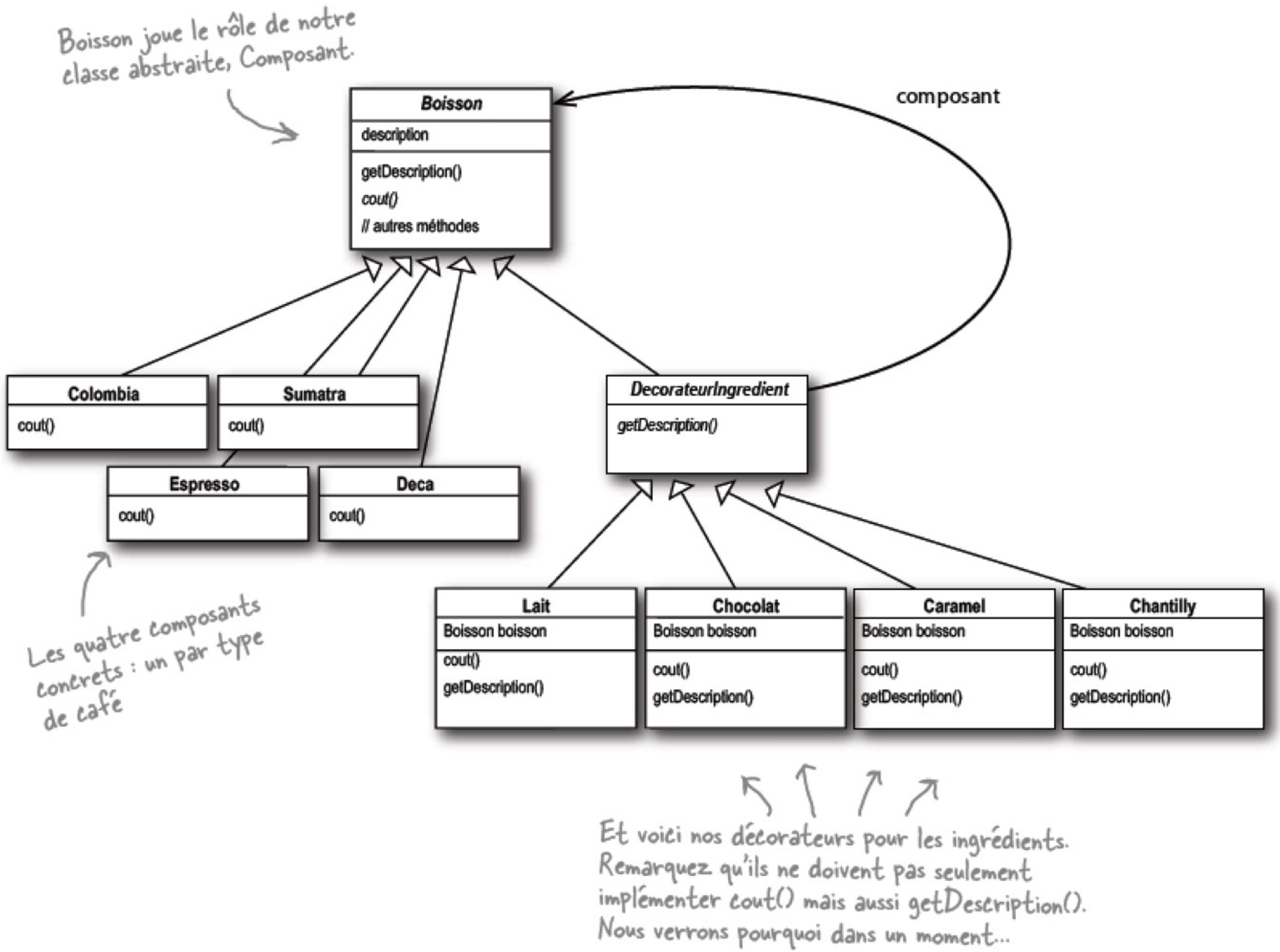
Augmentation du prix des ingrédients oblige à **modifier le code existant.**
Quid des nouveaux ingrédients?
Quid des nouvelles boissons?

Les classes doivent être ouvertes à l'extension, mais fermées à la modification.



Le pattern **Décorateur** attache dynamiquement des responsabilités/fonctionnalités supplémentaires à un objet. Il fournit une alternative souple à l'héritage, pour étendre les fonctionnalités.




Boisson joue le rôle de notre classe abstraite, Composant.
composant
Boisson
description
getDescription()
cout()
// autres méthodes
Colombia
cout()
Sumatra
cout()
Espresso
cout()
Deca
cout()
DecorateurIngredient
getDescription()
Les quatre composants concrets : un par type de café
Lait
Boisson boisson
cout()
getDescription()
Chocolat
Boisson boisson
cout()
getDescription()
Caramel
Boisson boisson
cout()
getDescription()
Chantilly
Boisson boisson
cout()
getDescription()
Et voici nos décorateurs pour les ingrédients. Remarquez qu'ils ne doivent pas seulement implémenter cout() mais aussi getDescription(). Nous verrons pourquoi dans un moment...

```
public abstract class Boisson {
       String description = "Boisson inconnue";

    public String getDescription() {
       return description;
    }

    public abstract double cout();
}
```

Boisson est une classe abstraite qui possède deux méthodes : getDescription() et cout().

getDescription a déjà été implémentée pour nous, mais nous devons implémenter cout() dans les sous-classes.

```
public class Colombia extends Boisson {
    public Colombia() {
        description = "Pur Colombia";
    }

    public double cout() {
        return .89;
    }
}
```

```
public abstract class DecorateurIngredient extends Boisson {
    public abstract String getDescription();
}
```

D'abord, comme elle doit être interchangeable avec une Boisson, nous étendons la classe Boisson.

Nous allons aussi faire en sorte que les ingrédients (décorateurs) réimplémentent tous la méthode getDescription(). Nous allons aussi voir cela dans une seconde...

Chocolat est un décorateur : nous étendons DecorateurIngredient.

Souvenez-vous que DecorateurIngredient étend Boisson.

```
public class Chocolat extends DecorateurIngredient {
   Boisson boisson;

   public Chocolat(Boisson boisson) {
      this.boisson = boisson;
   }

   public String getDescription() {
      return boisson.getDescription() + ", Chocolat";
   }

   public double cout() {
      return .20 + boisson.cout();
   }
}
```

Nous allons instancier Chocolat avec une référence à une Boisson en utilisant :

(1) Une variable d'instance pour contenir la boisson que nous enveloppons.

(2) Un moyen pour affecter à cette variable d'instance l'objet que nous enveloppons. Ici, nous allons transmettre la boisson que nous enveloppons au constructeur du décorateur.

La description ne doit pas comprendre seulement la boisson – disons « Sumatra » – mais aussi chaque ingrédient qui décore la boisson, par exemple, « Sumatra, Chocolat ». Nous allons donc déléguer à l'objet que nous décorons pour l'obtenir, puis ajouter « Chocolat » à la fin de cette description.

Nous devons maintenant calculer le coût de notre boisson avec Chocolat. Nous déléguons d'abord l'appel à l'objet que nous décorons pour qu'il calcule son coût. Puis nous ajoutons le coût de Chocolat au résultat.

```
public class StarbuzzCoffee {

    public static void main(String args[]) {
       Boisson boisson = new Espresso();
       System.out.println(boisson.getDescription()
           + " €" + boisson.cout());

        Boisson boisson2 = new Sumatra();
        boisson2 = new Chocolat(boisson2);
        boisson2 = new Chocolat(boisson2);
        boisson2 = new Chantilly(boisson2);
        System.out.println(boisson2.getDescription()
           + " €" + boisson2.cout());

        Boisson boisson3 = new Colombia();
        boisson3 = new Caramel(boisson3);
        boisson3 = new Chocolat(boisson3);
        boisson3 = new Chantilly(boisson3);
        System.out.println(boisson3.getDescription()
           + " €" + boisson3.cout());
    }
}
```

Commander un espresso, pas d'ingrédients et afficher sa description et son coût.

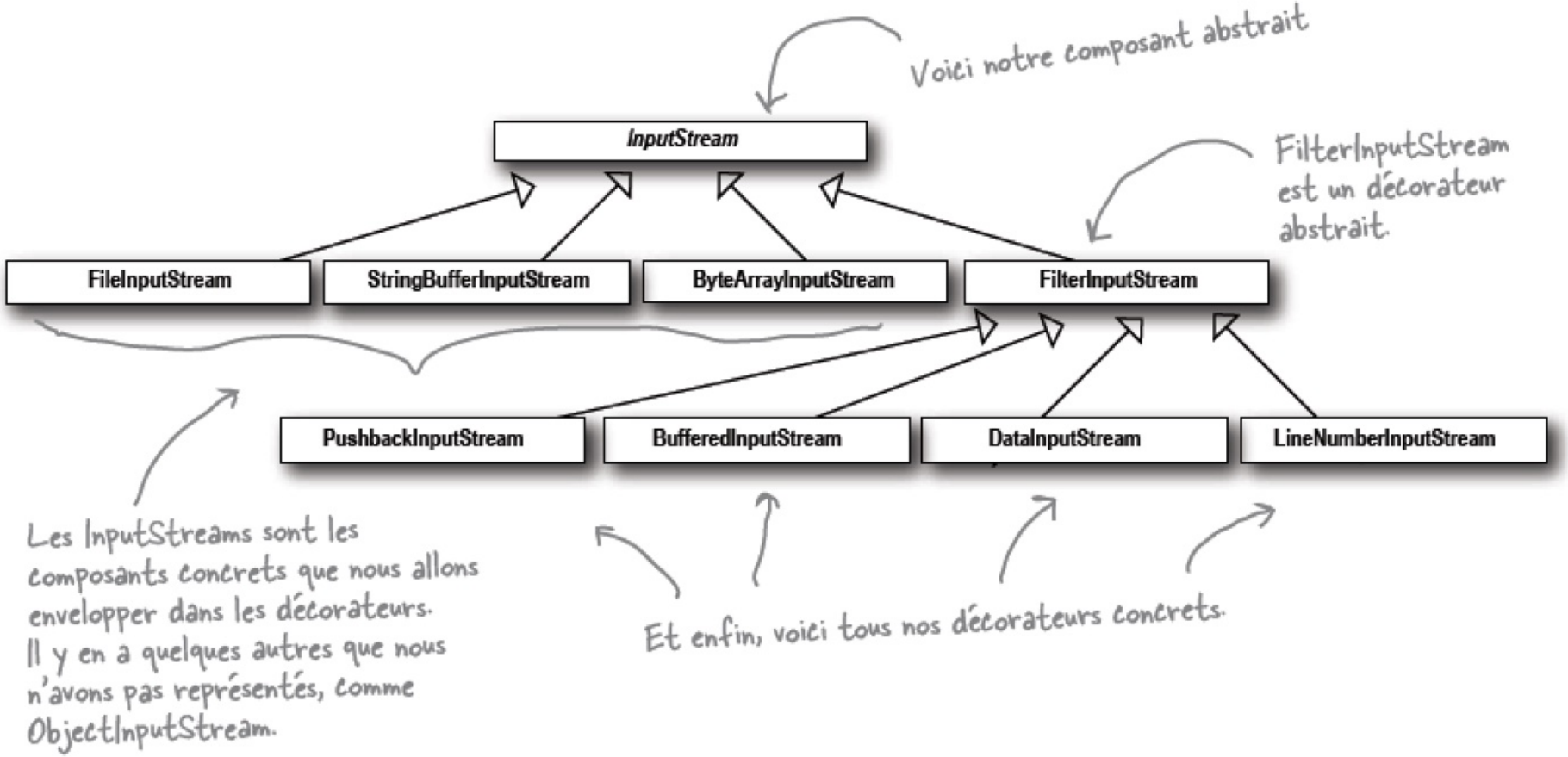
Créer un objet Sumatra.

L'envelopper dans un Chocolat.

L'envelopper dans un second Chocolat.

L'envelopper de Chantilly.

Enfin nous servir un Colombia avec Caramel, Chocolat et Chantilly.


Voici notre composant abstrait
InputStream
FilterInputStream est un décorateur abstrait.
FileInputStream
StringBufferInputStream
ByteArrayInputStream
FilterInputStream
PushbackInputStream
BufferedInputStream
DataInputStream
LineNumberInputStream
Les InputStreams sont les composants concrets que nous allons envelopper dans les décorateurs. Il y en a quelques autres que nous n'avons pas représentés, comme ObjectInputStream.
Et enfin, voici tous nos décorateurs concrets.

Adapter versus Decorator ?

Décorateur = Ajout de comportements aux opérations existantes. Interface non modifiée.

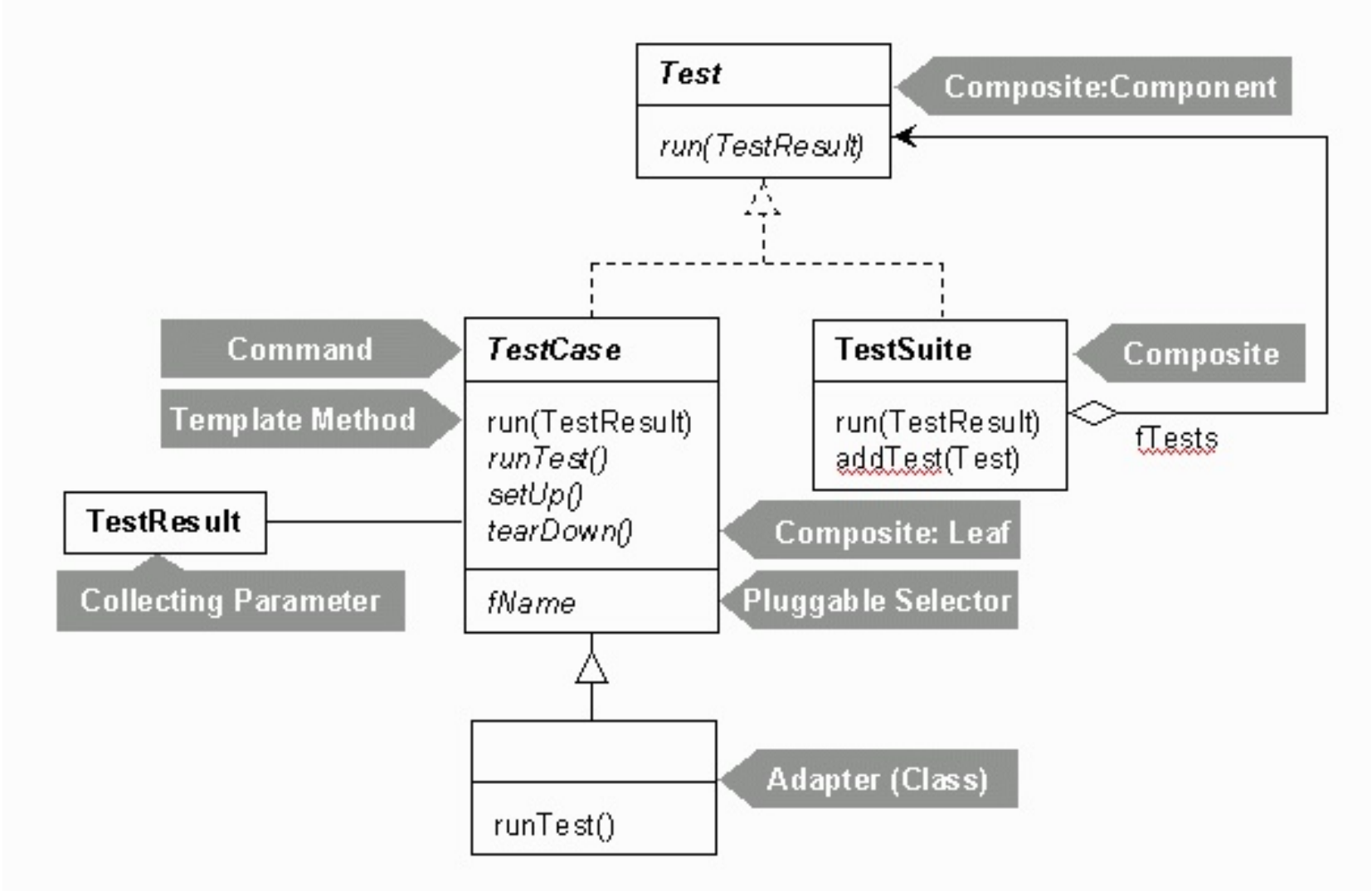
Adaptateur = Ajout d'interfaces à un objet

Conclusion

Key takeaways

- Design patterns offers standard de facto reusable solutions to recurrent « problems »
- « Problems » are usually corresponding to a lack of abstractions in the implementation language
  - And may vary according to the implementation languages
- « Solutions » are generic, and must be applied over the collaboration of domain concepts
  - But offer a common vocabulary

JUnit and… Design patterns



Worth reading!

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

JUnit and… Design patterns

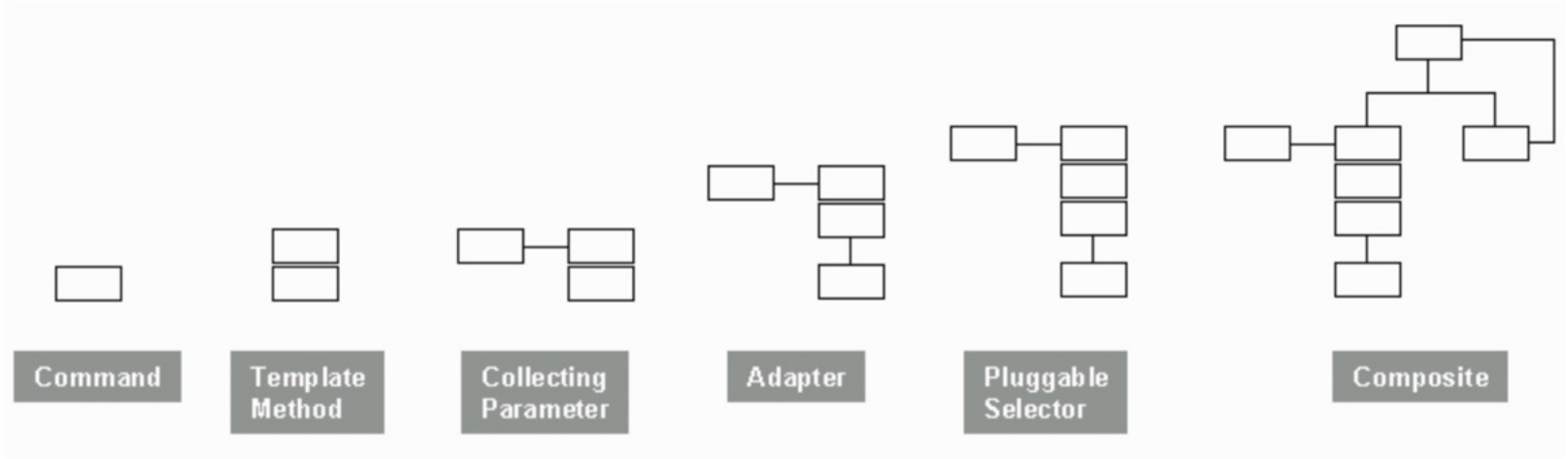


Figure 7: JUnit Pattern Storyboard

Worth reading!

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

Design Patterns from GOF

- C Abstract Factory
- S Adapter
- S Bridge
- C Builder
- B Chain of Responsibility
- B Command
- S Composite
- S Decorator
- S Facade
- C Factory Method
- S Flyweight
- B Interpreter
- B Iterator
- B Mediator
- B Memento
- C Prototype
- S Proxy
- B Observer
- C Singleton
- B State
- B Strategy
- B Template Method
- B Visitor

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

a new categorization

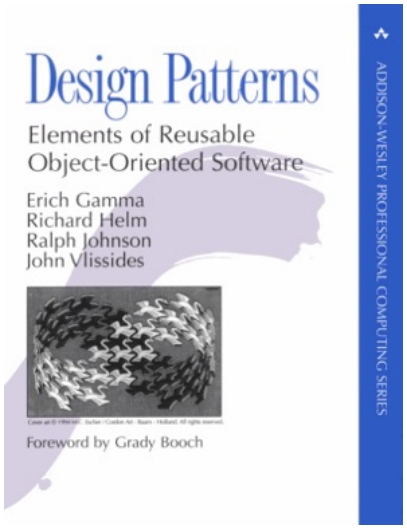
- **Core**
  - Composite
  - Strategy
  - State
  - Command
  - Iterator
  - Proxy
  - Template Method
  - Facade
  - *Null Object*

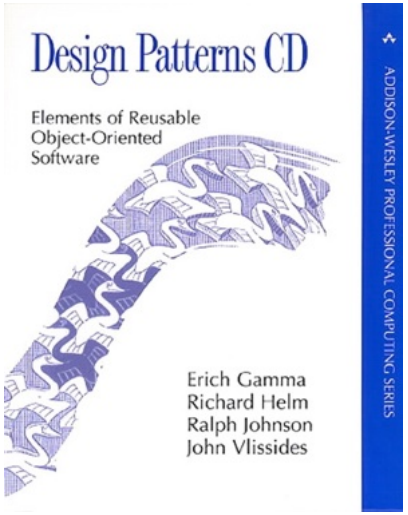
- **Creational**
  - Factory method
  - Prototype
  - Builder
  - *Dependency Injection*
- **Peripheral**
  - Abstract Factory (peripheral)
  - Memento
  - Chain of responsibility
  - Bridge
  - Visitor
  - *Type Object*
  - Decorator
  - Mediator
  - Singleton
  - *Extension Objects*
- **Other** (Compound)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

the patterns the students should learn

lean on demand

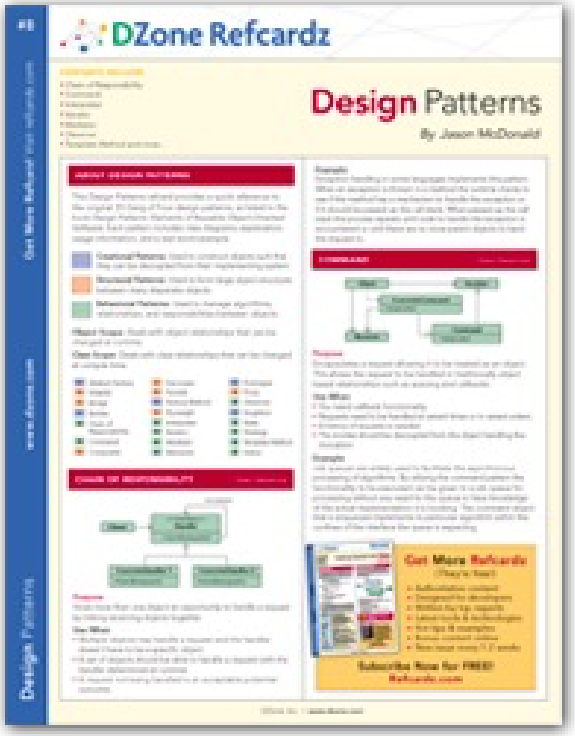
References


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

References



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