



Le génie pour l'industrie

# LOG121

## Conception orientée objet

Patrons Observateur et Stratégie  
Architecture MVC

Enseignante: Souad Hadjres

- Patron Observateur
- Introduction à l'architecture MVC
- Patron Stratégie

# Exemple de problème de conception

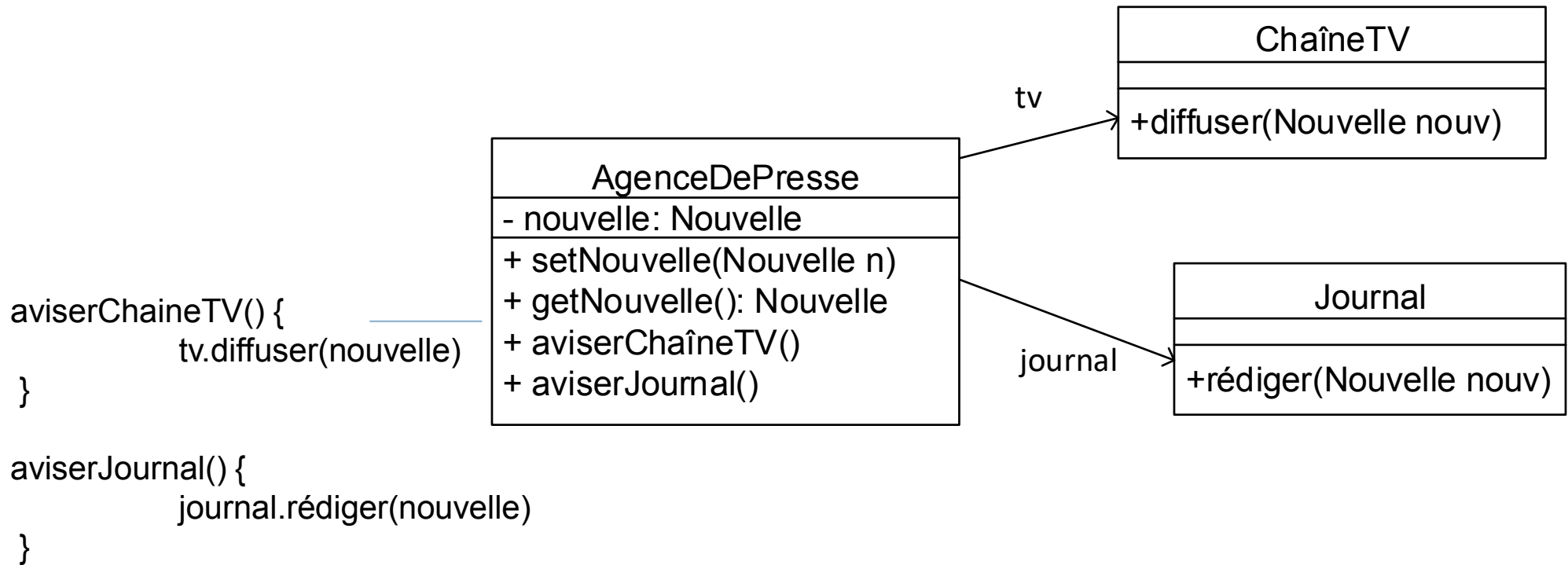
3

- Une agence de presse fournit les nouvelles à plusieurs médias
- Un média peut être une chaîne de radio, de télévision, un journal, etc.
- Un média a besoin de certaines parties des nouvelles

# Exemple de problème de conception

4

## □ Une conception simple .....



## □ Quel est le problème avec cette conception?

# Exemple de problème de conception

5

## □ Les problèmes

- La classe AgenceDePresse va être couplée à toutes les classes qui ont besoin des nouvelles
  - Ce couplage augmente à chaque ajout de nouvelle classe de média
- Elle doit aussi connaître quelles méthodes appeler de chaque classe de média
  - Quelle méthode et quelle partie de la nouvelle ou type de nouvelle envoyer à chaque média

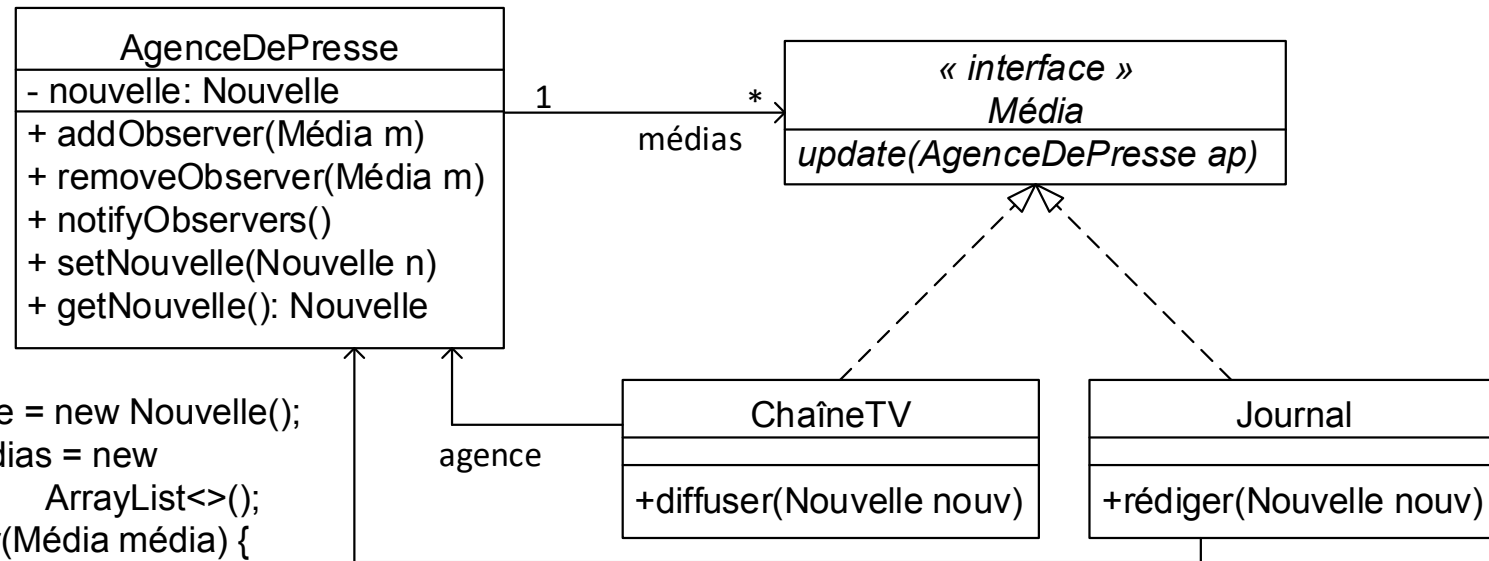
# Solution au problème

6

- Découpler la classe AgenceDePresse des classes de média
  - ▣ On va introduire une interface commune aux médias
  - ▣ Les médias s'abonnent auprès de l'agence
  - ▣ L'agence de presse doit avertir les médias abonnés lorsqu'il y a une nouvelle 'Nouvelle'
  - ▣ L'agence de presse ne connaît rien des médias à part qu'ils doivent être notifiées quand il y a des nouvelles
  - ▣ Le seul lien entre l'agence de presse et les médias est l'abonnement

# Solution au problème

7



```

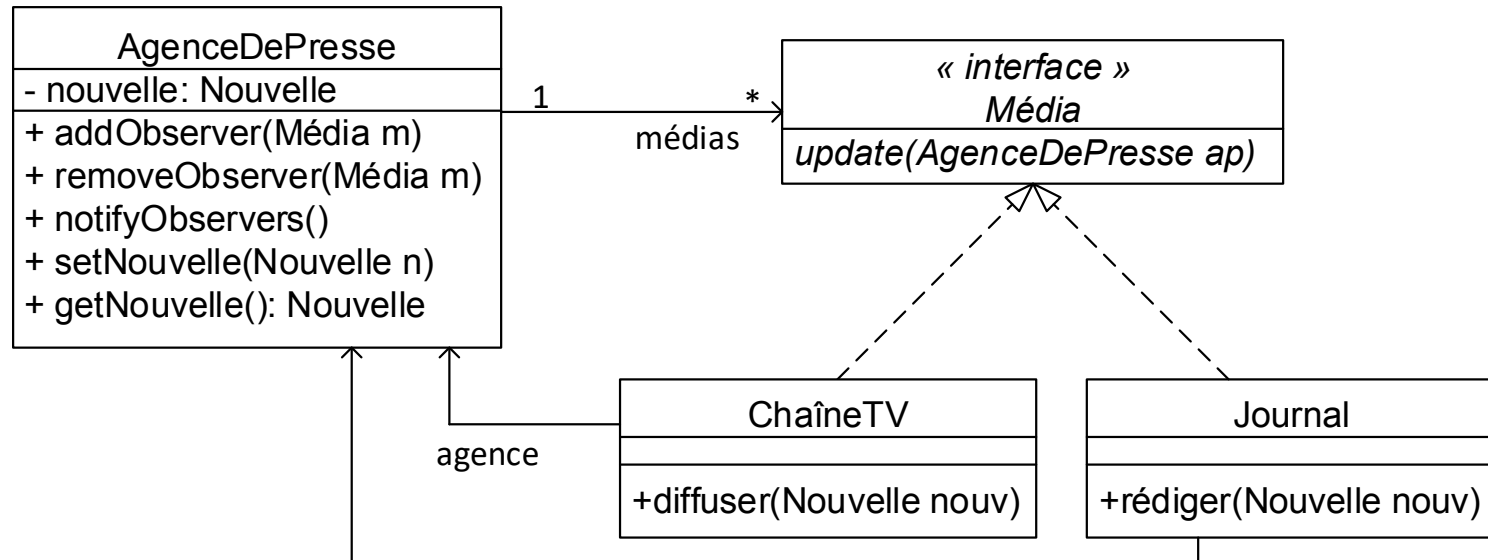
public class AgenceDePresse {
    private Nouvelle nouvelle = new Nouvelle();
    private List<Média> médias = new
        ArrayList<>();
    public void addObserver(Média média) {
        this.médias.add(média);
    }
    public void removeObserver(Média média) {
        this.médias.remove(média);
    }
    public void notifyObservers(){
        for (Média média : this.médias) {
            média.update(this);
        }
    }
    public Nouvelle getNouvelle() {
        return nouvelle;
    }
    public void setNouvelle(Nouvelle nouvelle) {
        this.nouvelle = nouvelle;
        this.notifyObservers();
    }
}
    
```

```

public class ChaîneTV implements Média {
    private AgenceDePresse agence;
    public void update(AgenceDePresse agence1) {
        agence = agence1;
        diffuser(agence.getNouvelle());
    }
    public void diffuser(Nouvelle : nouvelle){
        .....
    }
}
    
```

# Solution au problème

8



```
public class ClasseDeTest{
    public static void main (String[] args) {
        AgenceDePresse agence = new AgenceDePresse();
        Média CBC = new ChaîneTV();
        Média JDM = new Journal();
        agence.addObserver(CBC);
        agence.addObserver(JDM);
        Nouvelle dernièreHeure = new Nouvelle("Tremblement de terre au large du pacifique");
        agence.setNouvelle(dernièreHeure); // déclenche la notification
    }
}
```



# Solution au problème

9

- ❑ Le même principe s'applique dans plusieurs applications
- ❑ Par exemple aux boutons dans Java Swing
  - ▣ Un bouton avertit ses "abonnés" (*listeners*) lorsqu'il y a une action sur le bouton
  - ▣ Des "*Action listeners*" s'abonnent à un bouton afin d'être avertis
- ❑ Généralisation du principe en un patron
  - ▣ Des *observateurs* s'abonnent à un *sujet* qui les intéresse

# Le patron Observateur

10

## □ Contexte:

- ▣ Un objet, nommé le sujet, est une source d'évènements
- ▣ Un ou plusieurs observateurs s'intéressent à ces évènements et voudraient être avertis à l'arrivée de ces événements
- ▣ On ne connaît pas ces observateurs à priori
- ▣ On ne veut pas créer un fort couplage entre le sujet et ces observateurs

# Le patron Observateur

11

## □ Solution

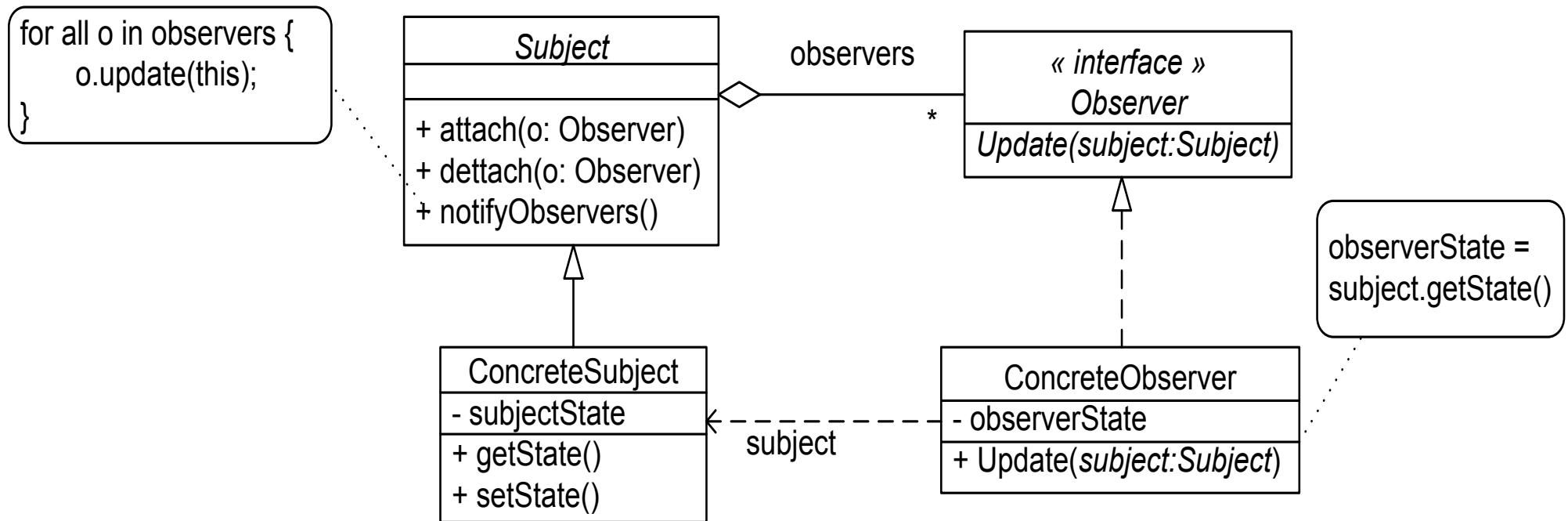
- ▣ Définir un type interface observateur (*Observer*).  
Tout observateur concret l'implémente.
- ▣ Le sujet gère une collection d'observateurs.
- ▣ Le sujet fournit des méthodes pour ajouter ou enlever des observateurs.
- ▣ Lorsqu'un évènement arrive, le sujet avertit tous les observateurs dans la collection.

# Le patron Observateur

12

## □ La structure générique du patron selon GoF

- C'est cette structure que nous allons utiliser dans le cours

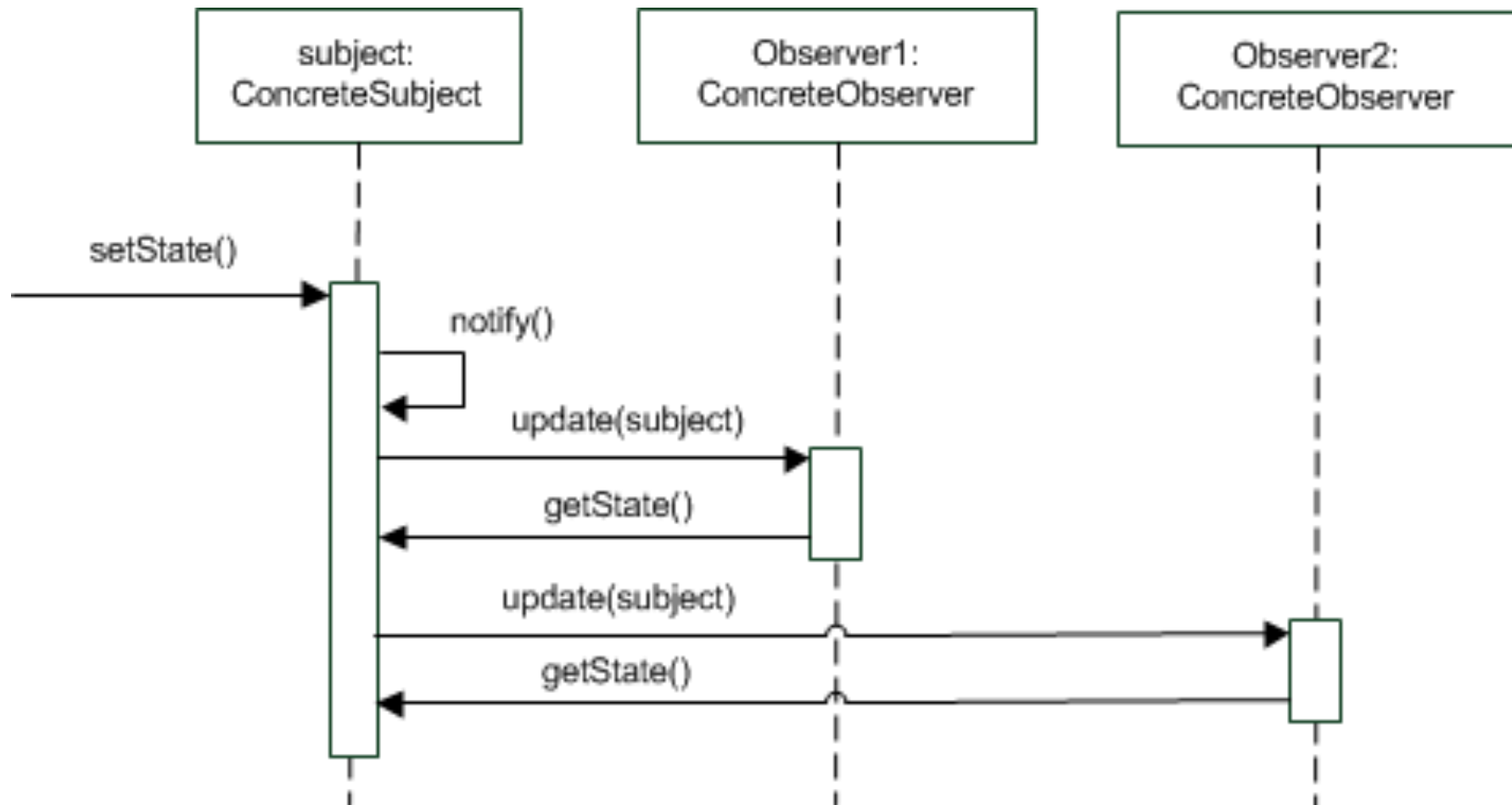


Attention: Ce diagramme est générique. Dans la pratique, `subjectState` sera représenté par un ou plusieurs attributs de la classe jouant le rôle de sujet concret dans votre application. Cette remarque est aussi valable pour `observerState`.

# Le patron Observateur

13

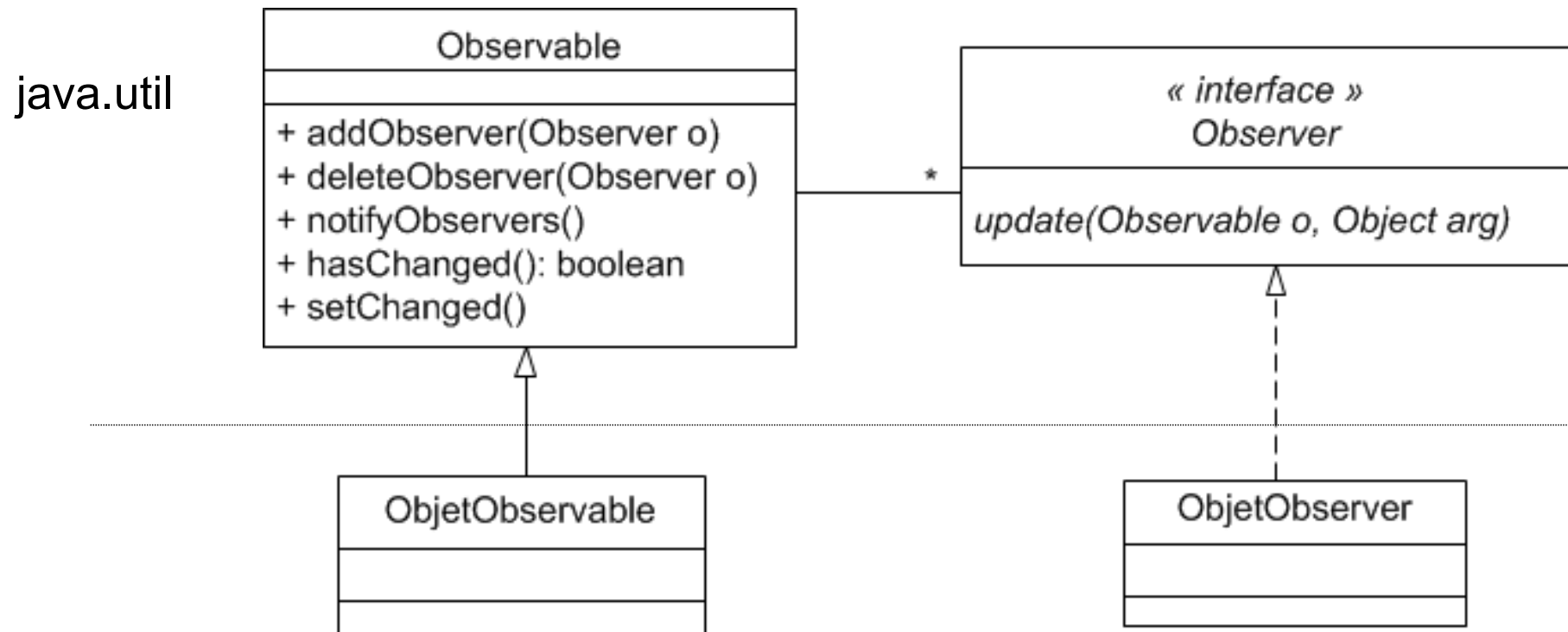
## □ Les interactions des entités du patron



# Le patron Observateur

14

## □ L'observateur dans Java



# Le patron Observateur

15

- La structure du patron dans le livre de Horstman a été simplifiée
  - ▣ Nous ne l'utiliserons pas
- Le modèle de délégation d'événements dans Java est une forme spécialisée du patron observateur

<u>Nom dans le patron de conception</u>	<u>Nom dans l'exemple avec les Boutons Swing</u>
Subject	JButton
Observer	ActionListener
ConcreteObserver	la classe implémentant l'interface ActionListener
attach()	addActionListener()
update()	actionPerformed(ActionEvent e)

- Patron Observateur
- Introduction à l'architecture MVC
- Patron Stratégie



# Différentes vues des mêmes données

17

- Certaines applications ont plusieurs vues sur le même ensemble de données
- Lorsqu'on modifie les données, les vues doivent être mises à jour automatiquement
- Exemple: un système de vote

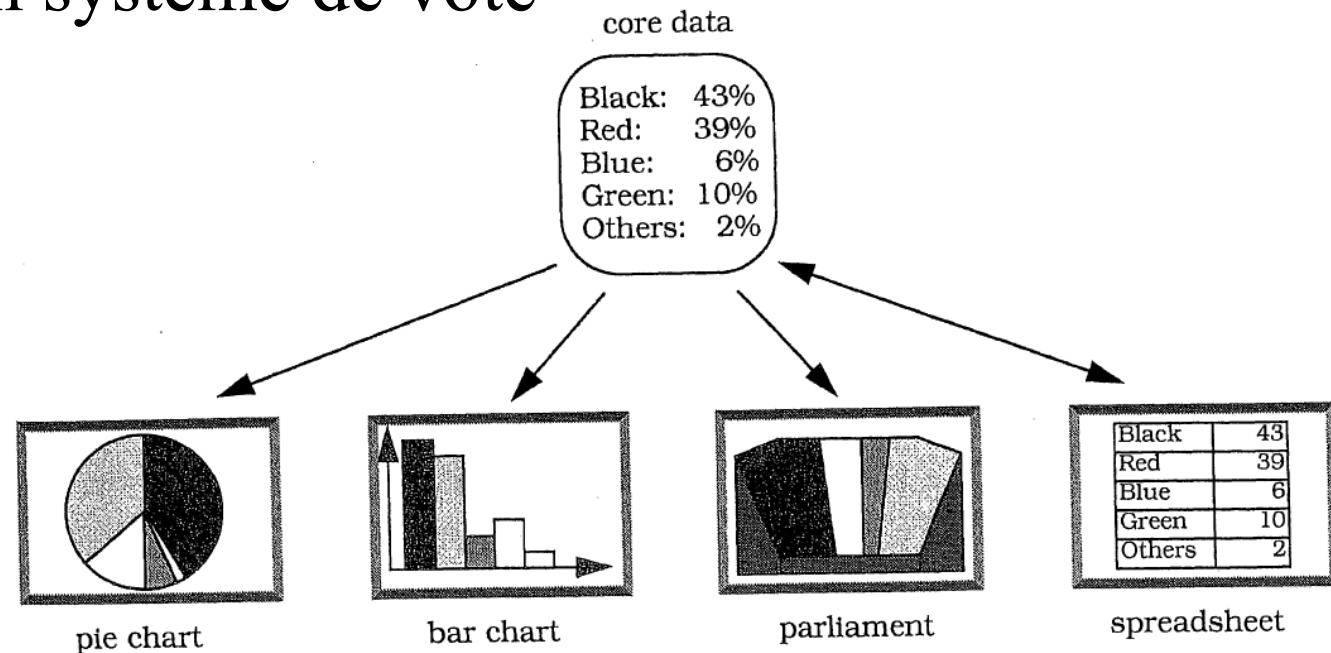


Figure extraite de « Pattern-Oriented Software Architecture, A System of Patterns », Buschmann et al., 1996

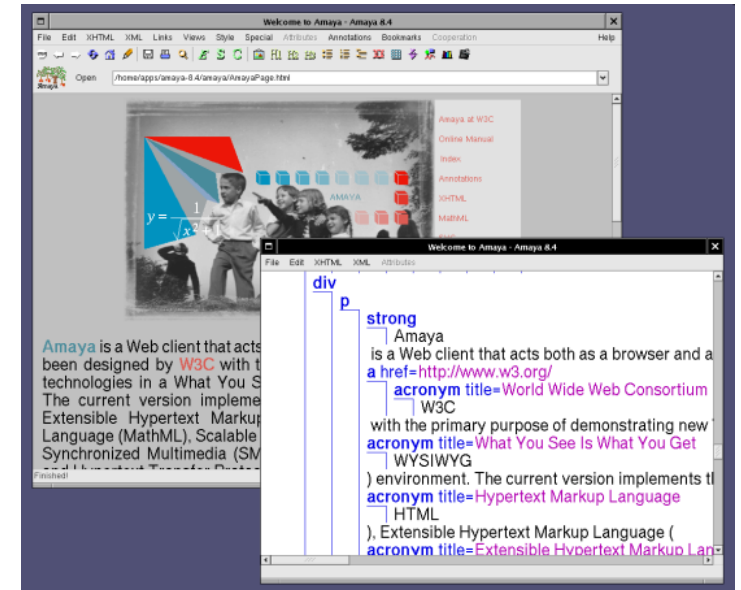
# Différentes vues des mêmes données

18

- Certaines applications ont différentes vues *modifiables* sur le même ensemble de données

- Exemple: éditeur HTML

- vue telle-quelle (WYSIWYG)
- vue source



- Lorsqu'on modifie l'une des vues, les autres doivent être mises à jour automatiquement et instantanément

# Exigences d'une application interactive

19

- La même information peut être présentée différemment aux utilisateurs
- L'affichage et le comportement de l'application doit immédiatement refléter les manipulations faites sur les données
- Les changements des interfaces utilisateurs (GUI\*) doivent être faciles
  - ▣ Les interfaces GUI sont susceptibles d'évoluer
  - ▣ Changer le « look and feel » ne devrait pas affecter le noyau de l'application

\* GUI: Graphical User Interface

# Architecture Modèle/Vue/ Contrôleur

20

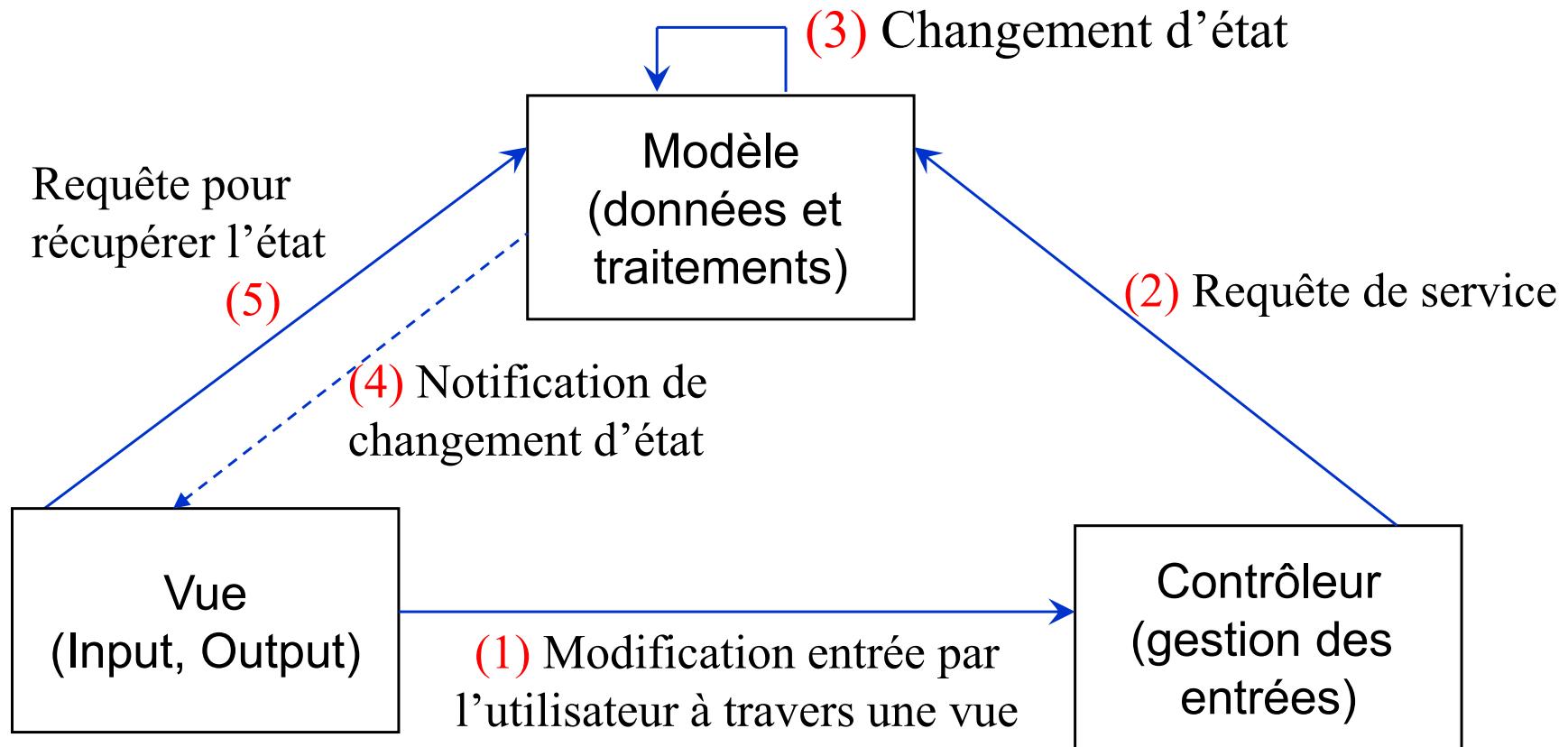
- MVC (Model-View-Controller) propose de diviser une application en trois parties
  - ▣ **Modèle:** encapsule les données et les fonctions noyau de l'application.
  - ▣ **Vues:** une vue présente les données du modèle. Une vue correspond aussi à une interface GUI à travers laquelle l'utilisateur déclenche des actions.
  - ▣ **Contrôleurs:** un contrôleur est associé à chaque vue. Il encapsule les actions déclenchées à travers la vue.

# Architecture Modèle/Vue/ Contrôleur

21

## ■ Fonctionnement

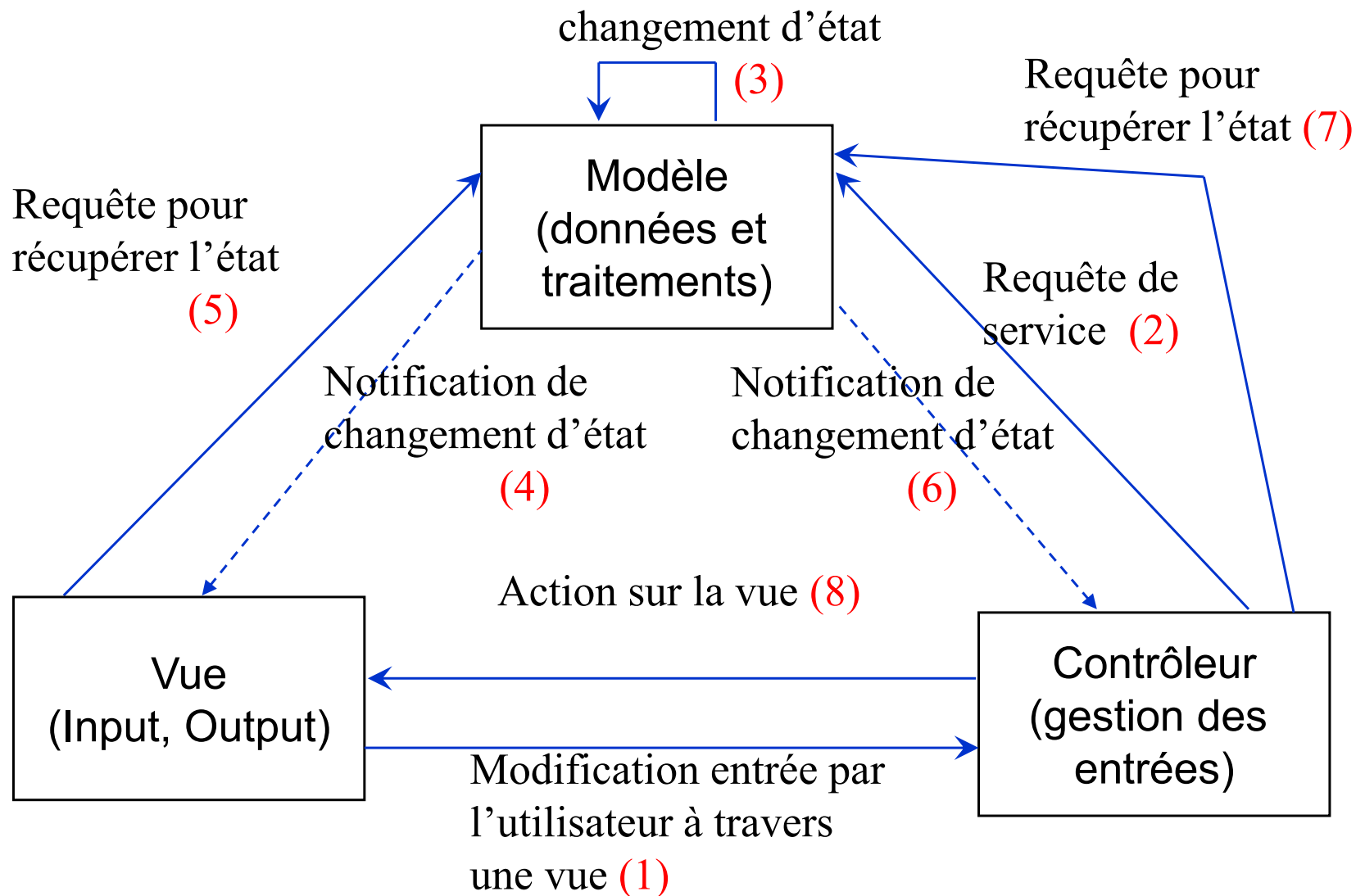
- Les vues et les contrôleurs modifient le modèle
- Le modèle notifie les vues des changements
- Les vues se mettent à jour en conséquence



# Architecture Modèle/Vue/ Contrôleur

22

## □ Fonctionnement plus élaboré

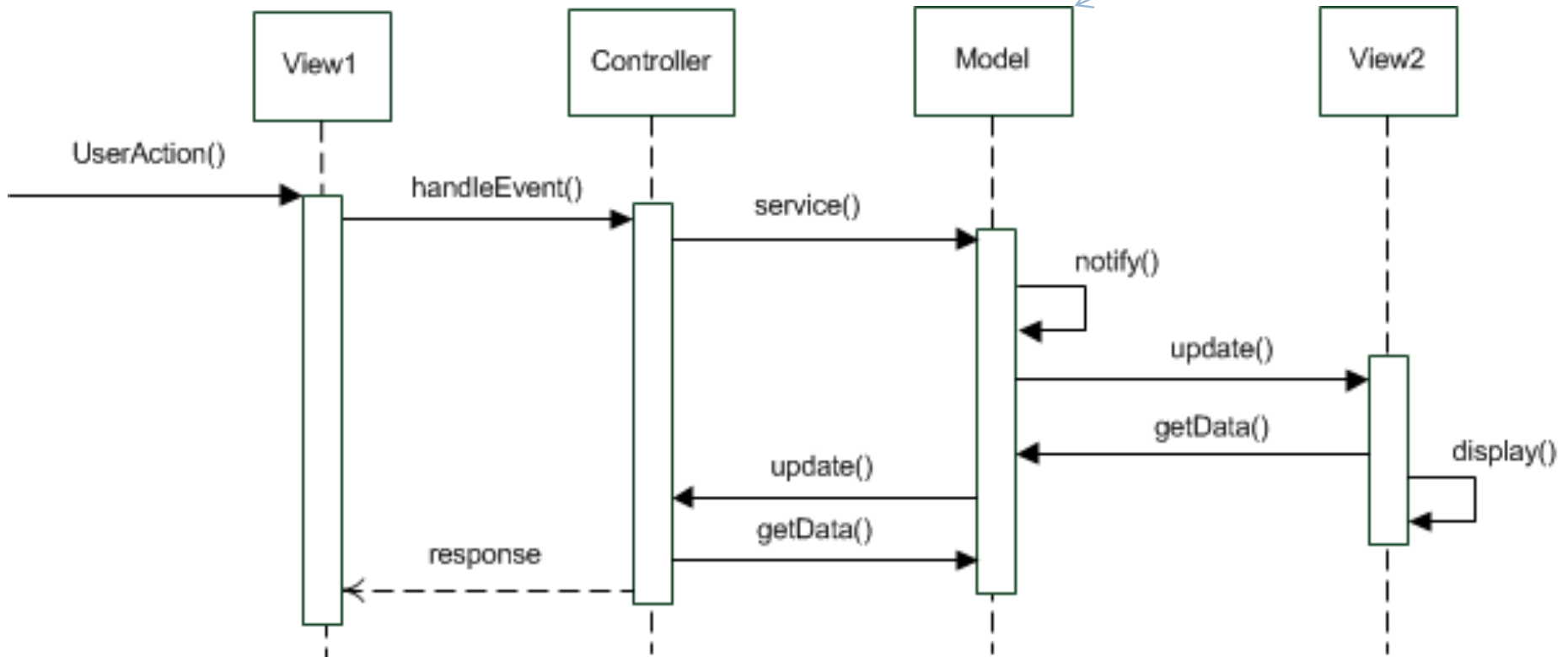


# Architecture Modèle/Vue/ Contrôleur

23

- ❑ Scénario où l'entrée faite par l'utilisateur change l'état du modèle\*

Ce diagramme est une simplification. Plusieurs classes du modèle peuvent être appelées



\* Diagramme adapté de «pattern-oriented software architecture», Buschman et al., 1996

## Le modèle

```
// The Model performs all the calculations needed
// and that is it. It doesn't know the View
// exists
```

```
public class CalculatorModel {
    // Holds the value of the sum of the numbers
    // entered in the view

    private int calculationValue;

    public void addTwoNumbers(int firstNumber, int secondNumber){
        calculationValue = firstNumber + secondNumber;
    }

    public int getCalculationValue(){
        return calculationValue;
    }
}
```

## Et pour tester le tout!

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

        CalculatorView theView = new CalculatorView();

        CalculatorModel theModel = new CalculatorModel();

        CalculatorController theController = new CalculatorController(theView, theModel);

        theView.setVisible(true);
    }
}
```

```
import java.awt.event.ActionEvent;
```

```
// The Controller coordinates interactions
// between the View and Model
```

```
public class CalculatorController {

    private CalculatorView theView;
    private CalculatorModel theModel;

    public CalculatorController(CalculatorView theView, CalculatorModel theModel) {
        this.theView = theView;
        this.theModel = theModel;

        // Tell the View that when ever the calculate button
        // is clicked to execute the actionPerformed method
        // in the CalculateListener inner class

        this.theView.addCalculateListener(new CalculateListener());
    }

    class CalculateListener implements ActionListener{

        public void actionPerformed(ActionEvent e) {
            int firstNumber, secondNumber = 0;

            // Surround interactions with the view with
            // a try block in case numbers weren't
            // properly entered

            try{
                firstNumber = theView.getFirstNumber();
                secondNumber = theView.getSecondNumber();
                theModel.addTwoNumbers(firstNumber, secondNumber);
                theView.setCalcSolution(theModel.getCalculationValue());
            }

            catch(NumberFormatException ex){
                System.out.println(ex);
                theView.displayErrorMessage("You Need to Enter 2 Integers");
            }
        }
    }
}
```

## Le contrôleur

```
// This is the View
import java.awt.event.ActionListener;
import javax.swing.*;
```

## La vue

```
public class CalculatorView extends JFrame{

    private JTextField firstNumber = new JTextField(10);
    private JLabel additionLabel = new JLabel("+");
    private JTextField secondNumber = new JTextField(10);
    private JButton calculateButton = new JButton("Calculate");
    private JTextField calcSolution = new JTextField(10);

    CalculatorView(){
        // Sets up the view and adds the components

        JPanel calcPanel = new JPanel();
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setSize(600, 200);

        calcPanel.add(firstNumber);
        calcPanel.add(additionLabel);
        calcPanel.add(secondNumber);
        calcPanel.add(calculateButton);
        calcPanel.add(calcSolution);
        calcSolution.setEditable(false);

        this.add(calcPanel);
        // this.add(calcPanel, BorderLayout.SOUTH);
    }

    public int getFirstNumber(){
        return Integer.parseInt(firstNumber.getText());
    }

    public int getSecondNumber(){
        return Integer.parseInt(secondNumber.getText());
    }

    public int getCalcSolution(){
        return Integer.parseInt(calcSolution.getText());
    }

    public void setCalcSolution(int solution){
        calcSolution.setText(Integer.toString(solution));
    }

    // If the calculateButton is clicked execute a method
    // in the Controller named actionPerformed

    void addCalculateListener(ActionListener listenForCalcButton){
        calculateButton.addActionListener(listenForCalcButton);
    }
}
```



- ❑ **MVC minimise le couplage** entre le modèle, les vues et les contrôleurs
  - ▣ Le modèle ne connaît rien des vues à part qu'elles doivent être notifiées des changements
  - ▣ Les vues ne connaissent pas les contrôleurs
- ➔ On peut facilement ajouter des vues

- Application du patron Observateur
  - ▣ Le modèle doit avertir les vues lorsqu'il arrive un évènement qui les intéresse
  - ▣ Le modèle ne connaît rien des vues à part qu'elles doivent être notifiées des changements
  - ▣ Les vues *s'abonnent* (s'enregistrent) auprès du modèle afin d'être averties
    - Le seul lien entre le modèle et les vues est cet abonnement
  - ▣ Un contrôleur peut s'abonner aussi au modèle lorsque son comportement dépend de l'état du modèle

## □ Attention:

- ▣ Le modèle est généralement implémenté par un ensemble de classes
  - Il englobe des classes du domaine d'affaire
- ▣ La vue aussi
  - Elle contient des classes implémentant une interface graphique (GUI)
- ▣ Il y a un contrôleur par vue

- Patron Observateur
- Introduction à l'architecture MVC
- Patron Stratégie

# Exemple de problème de conception

29

```
public class MaCollection {
    private ArrayList<String> elements = new ArrayList<String>();
    private String typeTri = new String("triRapide");

    public void addElement(String elt){
        elements.add(elt);
    }

    public void setSortType(String tri){
        typeTri = tri;
    }

    public void trier(){
        if (typeTri.matches("triRapide")){
            // implementer tri rapide ici
            //.....
            System.out.println("Tri rapide");
        }
        if (typeTri.matches("triFusion")){
            // implementer tri fusion ici
            //.....
            System.out.println("Tri fusion");
        }
        if (typeTri.matches("triInsertion")){
            // implementer tri par insertion ici
            //.....
            System.out.println("Tri par insertion");
        }
    }
}
```

```
public class MonApplication {
    public static void main(String[] args) {
        MaCollection capitales = new MaCollection();
        capitales.addElement("Ottawa");
        capitales.addElement("New York");
        capitales.addElement("Mexico");
        capitales.addElement("Paris");
        capitales.trier();
    }
}
```

# Exemple de problème de conception

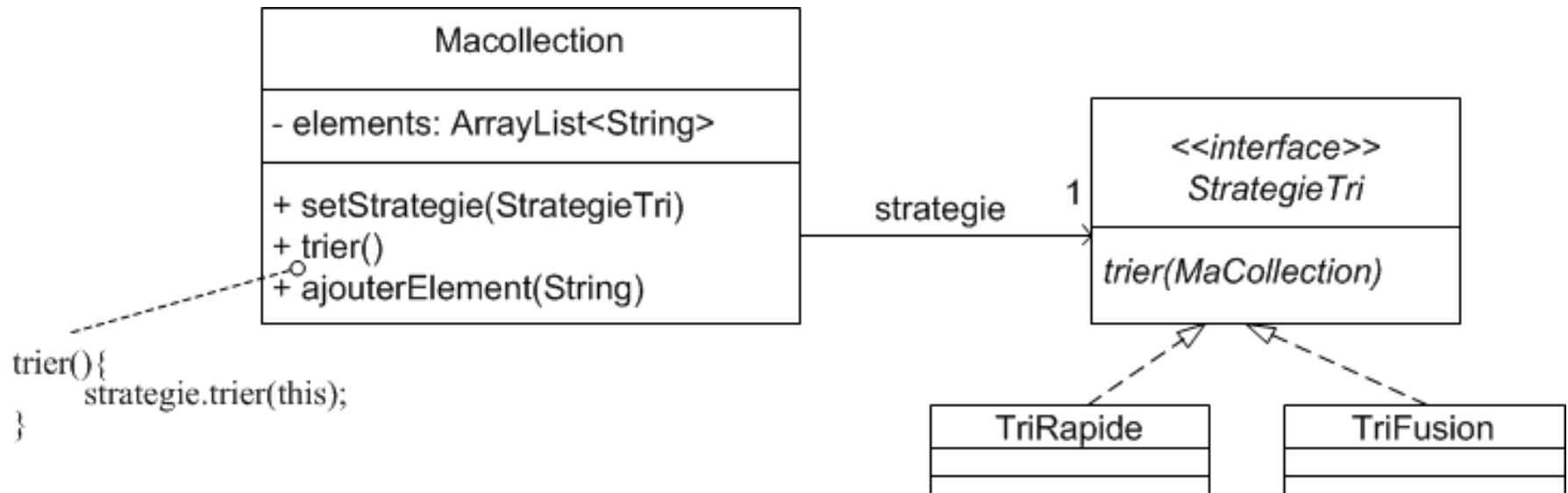
30

- ❑ Le code de la méthode concernée peut devenir long
- ❑ Il peut aussi devenir complexe à comprendre
- ❑ Difficulté de maintenance
- ❑ Difficulté d'ajouter de nouvelles variantes de l'algorithme de tri

- ❑ Encapsuler chaque variante de l'algorithme de tri dans un objet
- ❑ Définir une interface commune à ces objets
- ❑ La classe MaCollection utilise ces objets et permet d'en faire le choix à l'exécution

# Solution au problème

32



```
public class MaCollection {
    private ArrayList<String> elements = new ArrayList<String>();
    private StrategieTri strategie = new TriRapide();

    public void addElement(String elt){
        elements.add(elt);
    }

    public void setStrategy(StrategieTri currentStrategy){
        strategie = currentStrategy;
    }

    public void trier(){
        strategie.trier(this);
    }
}
```

- Il est facile d'ajouter d'autres algorithmes de tri
- La classe MaCollection est plus facile à maintenir



# Deuxième exemple

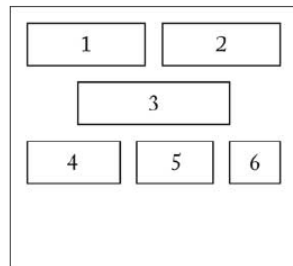
33

- L'interface utilisateur dans Java est construite en plaçant des composants « components » dans des containers
- Un Container a besoin de disposer les composants
- Swing n'utilise pas des coordonnées figées dans le code
  - ▣ permet de changer l'aspect et la convivialité ("look and feel")
  - ▣ permet l'internationalisation des chaînes de caractères
- Un gestionnaire de disposition (*Layout manager*) contrôle la façon de disposer les composants

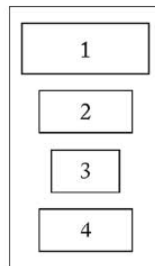
# Layout manager (Gestionnaire de disposition)

34

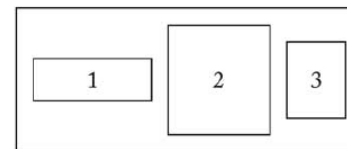
- *FlowLayout*: de gauche à droite, commence une nouvelle rangée lorsque celle courante est pleine
- *BoxLayout*: de gauche à droite ou de haut en bas
- *BorderLayout*: 5 zones, Center, North, South, East, West
- *GridLayout*: grille, tous les composants ont la même taille
- *GridBagLayout*: complexe, comme un tableau HTML



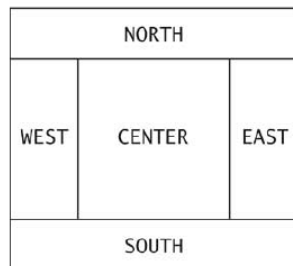
FlowLayout



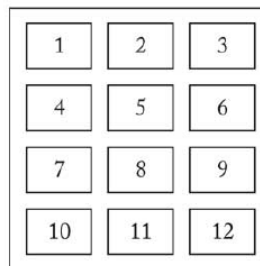
BoxLayout (vertical)



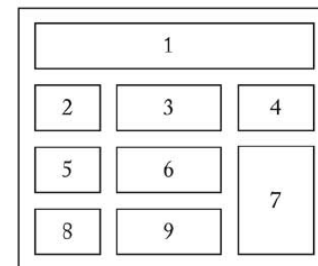
BoxLayout (horizontal)



BorderLayout



GridLayout



GridBagLayout

# Layout manager (Gestionnaire de disposition)

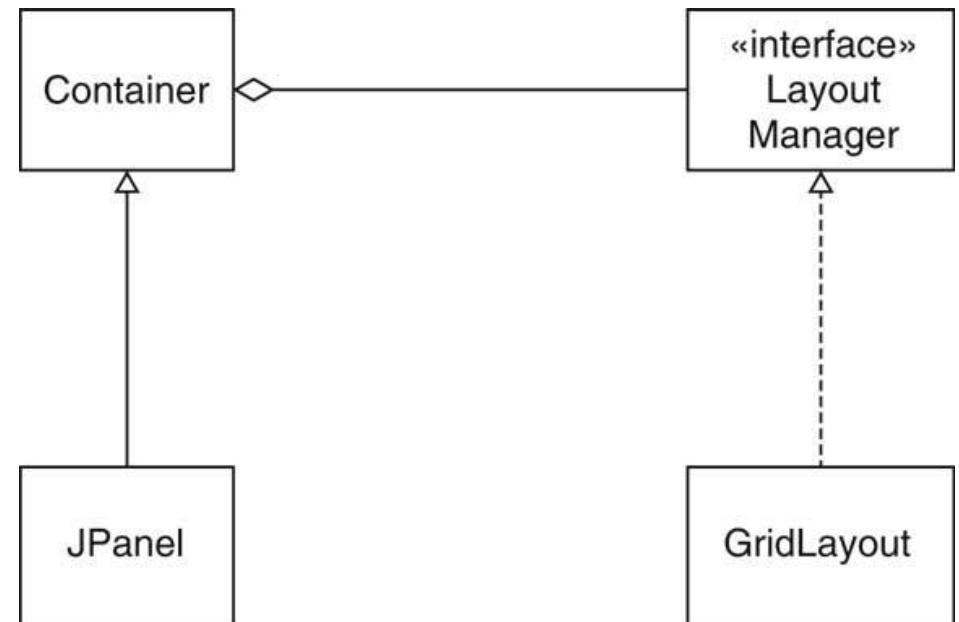
35

## □ Configurer le gestionnaire de disposition

```
JPanel keyPanel = new JPanel();  
keyPanel.setLayout(new GridLayout(4, 3));
```

## □ Ajouter des composants

```
for (int i = 0; i < 12; i++)  
    keyPanel.add(button[i]);
```



# Utilisation des gestionnaires de disposition

36

- Interface utilisateur pour simuler la classe Téléphone d'un système de boîte vocale [Ch5/mailgui/Telephone.java](#)

A screenshot of a Java Swing window titled "Telephone". The window contains a "Speaker:" section with the text "You have reached mailbox 12. Please leave a message now." Below this is a numeric keypad with buttons for digits 1-9, \*, 0, and #. Below the keypad is a "Microphone:" section with the text "Hello Fifi! This is Aramis. Are we still on for lunch today? Please call me back. Thanks!". At the bottom of the window are two buttons: "Send speech" and "Hangup".

1	2	3
4	5	6
7	8	9
*	0	#

un panneau avec  
BorderLayout

un panneau avec  
GridLayout

un panneau avec  
BorderLayout

un panneau avec  
BorderLayout

# Layout manager (Gestionnaire de disposition)

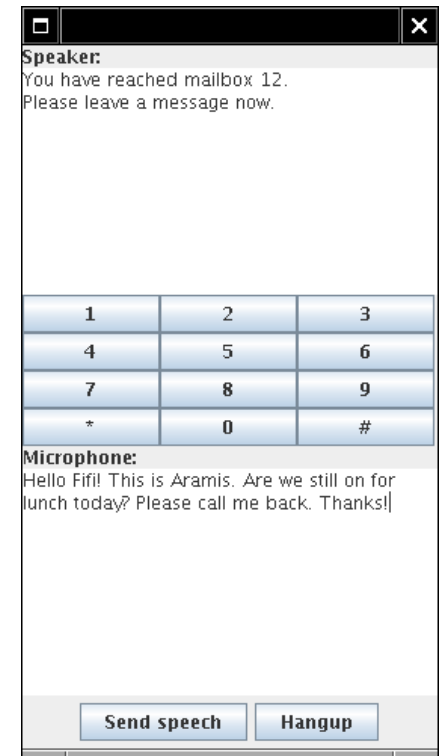
37

## □ Disposition des touches de composition

```
JPanel keyPanel = new JPanel();  
keyPanel.setLayout(new GridLayout(4, 3));  
for (int i = 0; i < 12; i++) {  
    JButton keyButton = new JButton(...);  
    keyPanel.add(keyButton);  
    keyButton.addActionListener(...);  
}
```

## □ Panneau du haut-parleur

```
JPanel speakerPanel = new JPanel();  
speakerPanel.setLayout(new BorderLayout());  
speakerPanel.add(new JLabel("Speaker:"), BorderLayout.NORTH);  
JTextArea speakerField = new JTextArea(10, 25);  
speakerPanel.add(speakerField, BorderLayout.CENTER);
```

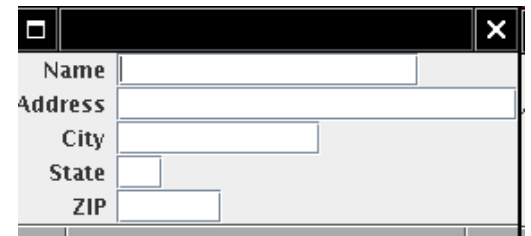


# Gestionnaire personnalisé de disposition

38

- Si on voulait définir sa propre façon de disposer les composants
  - ▣ Exemple: disposition sous forme de formulaire avec les composants impairs alignés à droite et les composants pairs alignés à gauche
- Il suffit d'implémenter l'interface `LayoutManager`

```
public interface LayoutManager
{
    void layoutContainer(Container parent);
    Dimension minimumLayoutSize(Container parent);
    Dimension preferredLayoutSize(Container parent);
    void addLayoutComponent(String name, Component comp);
    void removeLayoutComponent(Component comp);
}
```



[Ch5/layout/FormLayout.java](#)

[Ch5/layout/FormLayoutTester.java](#)

# Le patron Stratégie

39

- Cette flexibilité est une conséquence de l'encapsulation de la stratégie de disposition dans une classe à part
- La stratégie de disposition est interchangeable (*pluggable*)
- Un autre exemple d'encapsulation d'une stratégie ou algorithme: Comparators

```
Comparator<Country> comp = new CountryComparatorByName();
Collections.sort(countries, comp);
```
- Généralisation du principe dans le patron Stratégie

## □ Contexte

- ▣ Une classe peut nécessiter des variantes différentes d'un algorithme
- ▣ Les clients veulent parfois remplacer l'algorithme standard avec des versions personnalisées



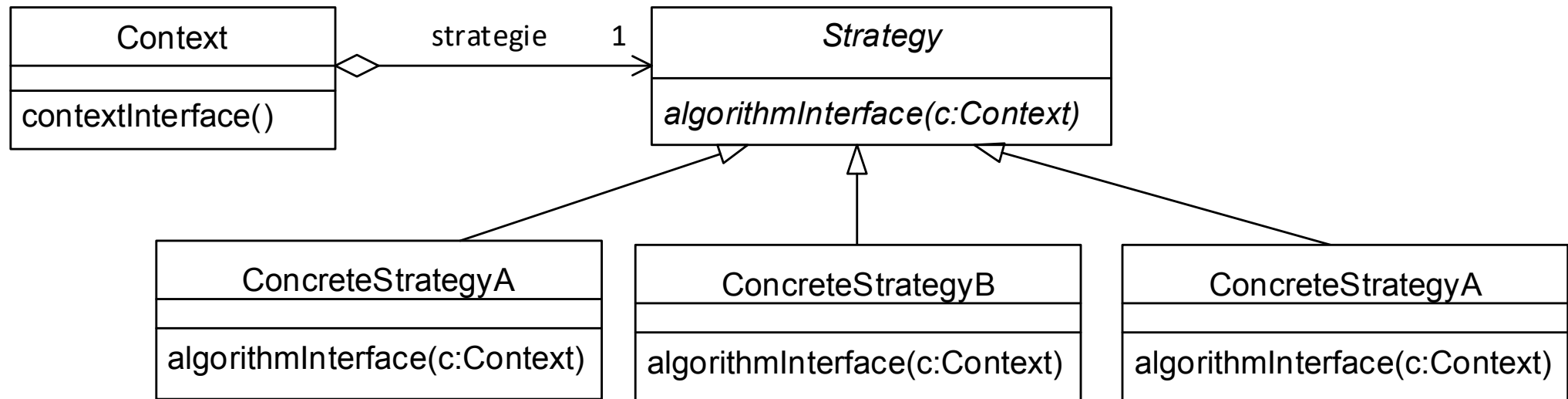
## □ Solution

- ▣ Définir des classes concrètes qui encapsulent les différents algorithmes. Ces classes sont appelées des stratégies.
- ▣ Les associer à une interface commune.
  - L'interface est une abstraction de l'algorithme
- ▣ Les clients peuvent fournir leurs propres stratégies
- ▣ Lorsque l'algorithme doit être exécuté, la classe contexte appelle les méthodes appropriées de l'objet stratégie

# Le patron Stratégie

42

## □ La structure générique du patron selon GoF



### Nom dans le patron de conception

Context  
contextInterface()  
Strategy  
ConcreteStrategyA  
algorithmInterface(c:Context)

### Nom dans l'exemple de tri

MaCollection  
trier()  
StrategieTri  
TriRapide  
trier(c:MaCollection)

- Quels sont les avantages et inconvénients du patron Stratégie?

*- flexibilité*

# Le patron Stratégie

44

## □ Avantages

- Plus de flexibilité quant à l'ajout d'autres stratégies et à leurs changements indépendamment du contexte.
- L'interface Stratégie permet de factoriser les fonctionnalités communes des algorithmes.

## □ Inconvénients

- Les clients doivent être informés des différentes stratégies disponibles.
- Prolifération du nombre d'objets.

- Exemples d'application
  - ▣ Calcul du prix de vente pour différents clients et à différents moments : réduction, promotion occasionnelle, réduction permanente pour personnes âgées, programme de fidélisation par cumul de points...
  - ▣ Compression d'un fichier avec différents formats.