



Le génie pour l'industrie

LOG121

Conception orientée objet

Groupe 3
Hiver 2020

Patrons Observateur et Stratégie
Architecture MVC

Chargé de cours: Cédric St-Onge

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

Différentes vues des mêmes données

3

- Certaines applications ont plusieurs vues sur le même ensemble de données
- Lorsqu'on modifie les données, les vues doivent être mises à jour automatiquement
- Exemple1: un système de vote

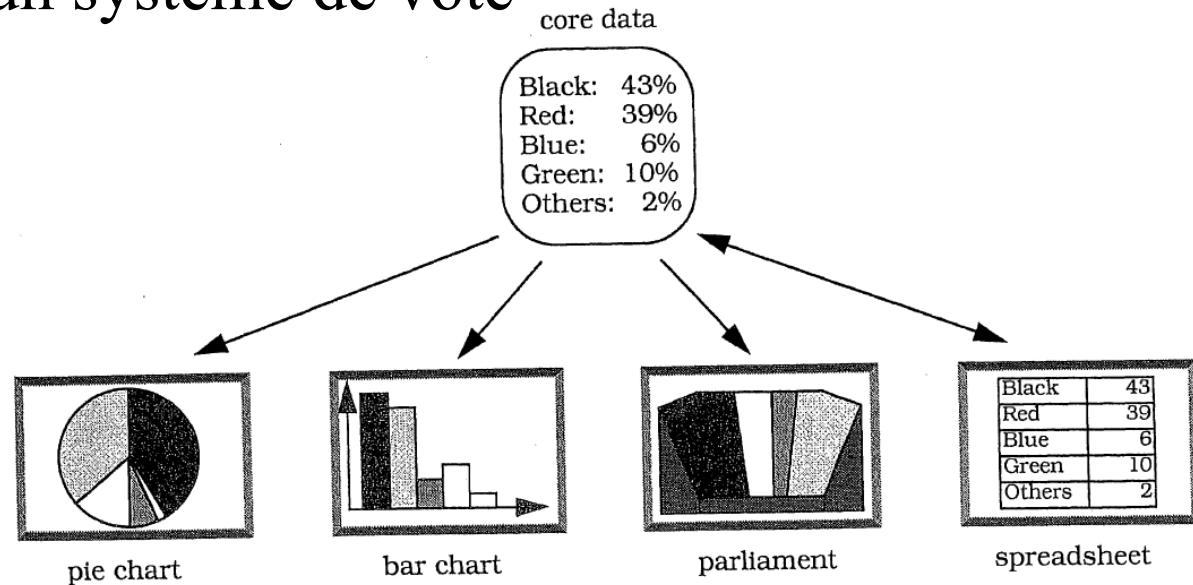


Figure extraite de « Pattern-Oriented Software Architecture, A System of Patterns », Buschmann et al., 1996

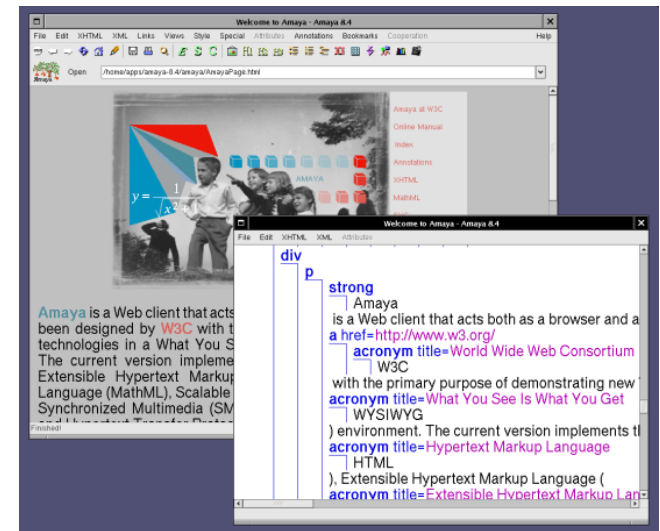
Différentes vues des mêmes données

4

- 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 mises à jour automatiquement et instantanément

Exigences d'une application interactive

5

- 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 doivent être faciles
 - ▣ Les interfaces sont susceptibles d'évoluer
 - ▣ Changer le « look and feel » ne devrait pas affecter le noyau de l'application

Architecture Modèle/Vue/ Contrôleur

6

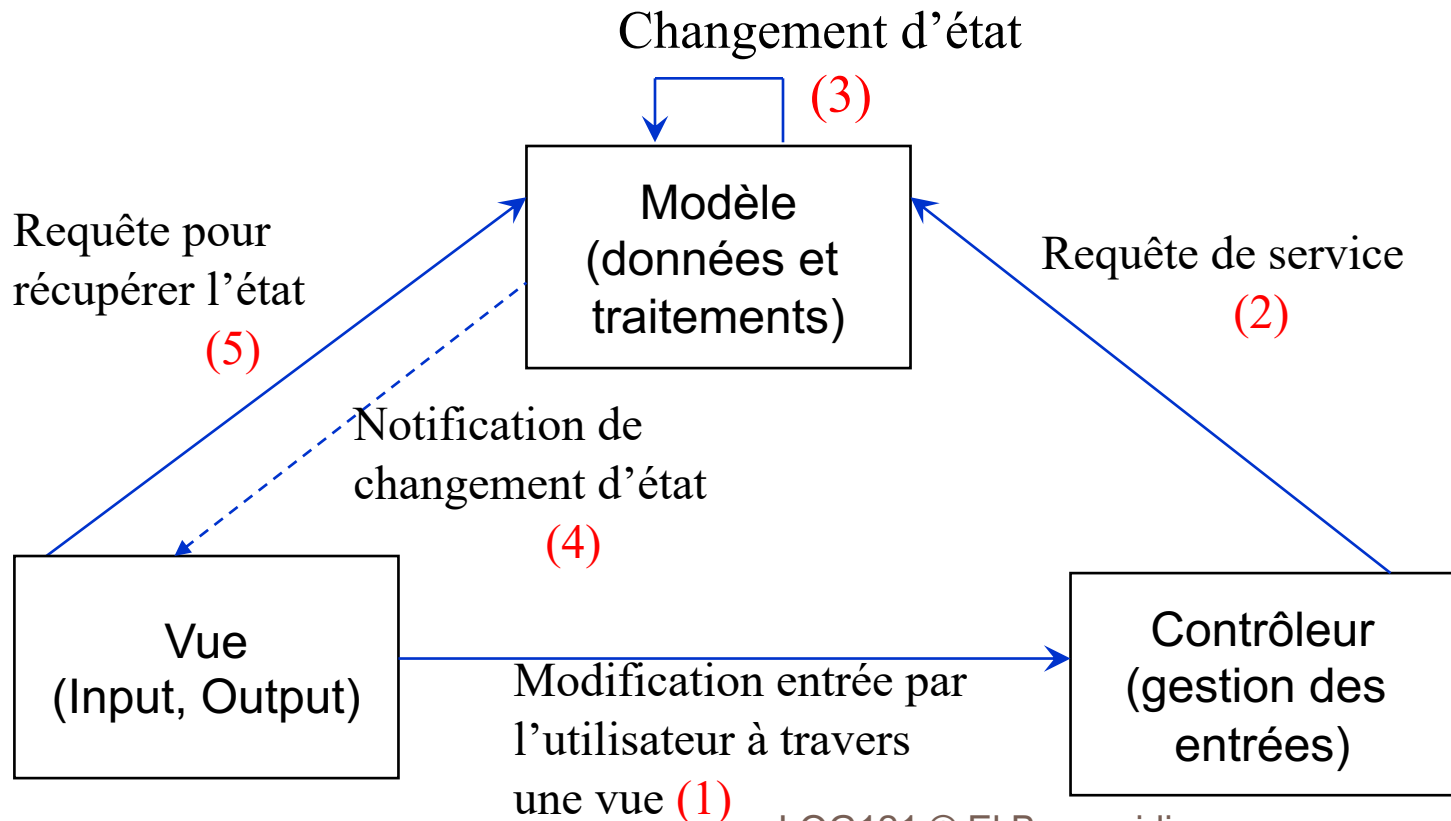
- MVC (Model-View-Controller) propose de diviser une application en trois parties
 - ▣ **Modèle:** encapsule les données et les fonctions noyau de l'application.
 - ▣ **Vues:** une vue présente les données du modèle. Une vue correspond aussi à une interface à travers laquelle l'utilisateur déclenche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

Architecture Modèle/Vue/ Contrôleur

7

■ Fonctionnement

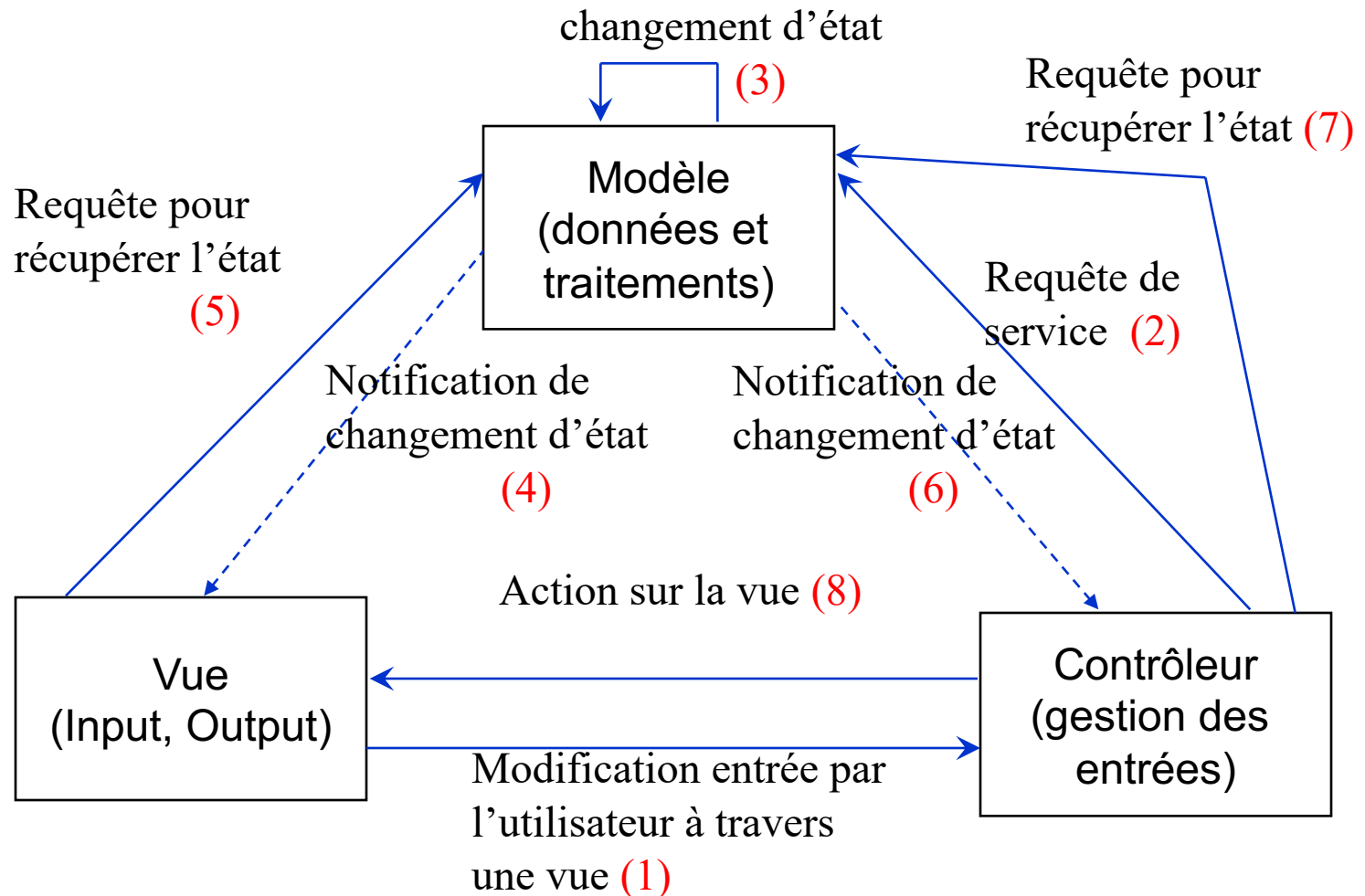
- Les vues et les contrôleurs modifient le modèle
- Le modèle informe les vues des changements
- Les vues se mettent à jour en conséquence



Architecture Modèle/Vue/ Contrôleur

8

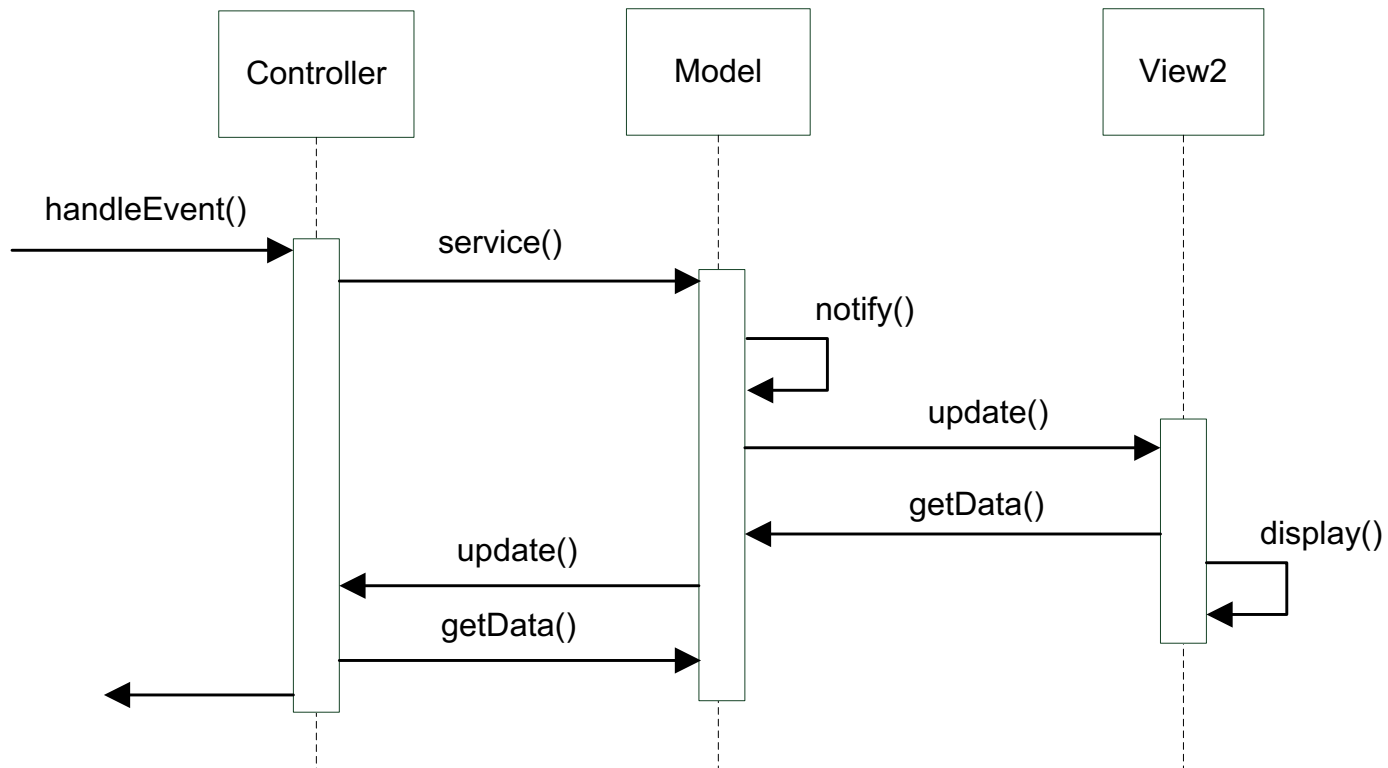
□ Fonctionnement plus élaboré



Architecture Modèle/Vue/ Contrôleur

9

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



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

- **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

□ 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
- ▣ Il y a un contrôleur par vue

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

- Dans l'architecture MVC
 - ▣ 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'attachent / s'enregistrent) au 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

Le principe

14

- 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

15

□ 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

16

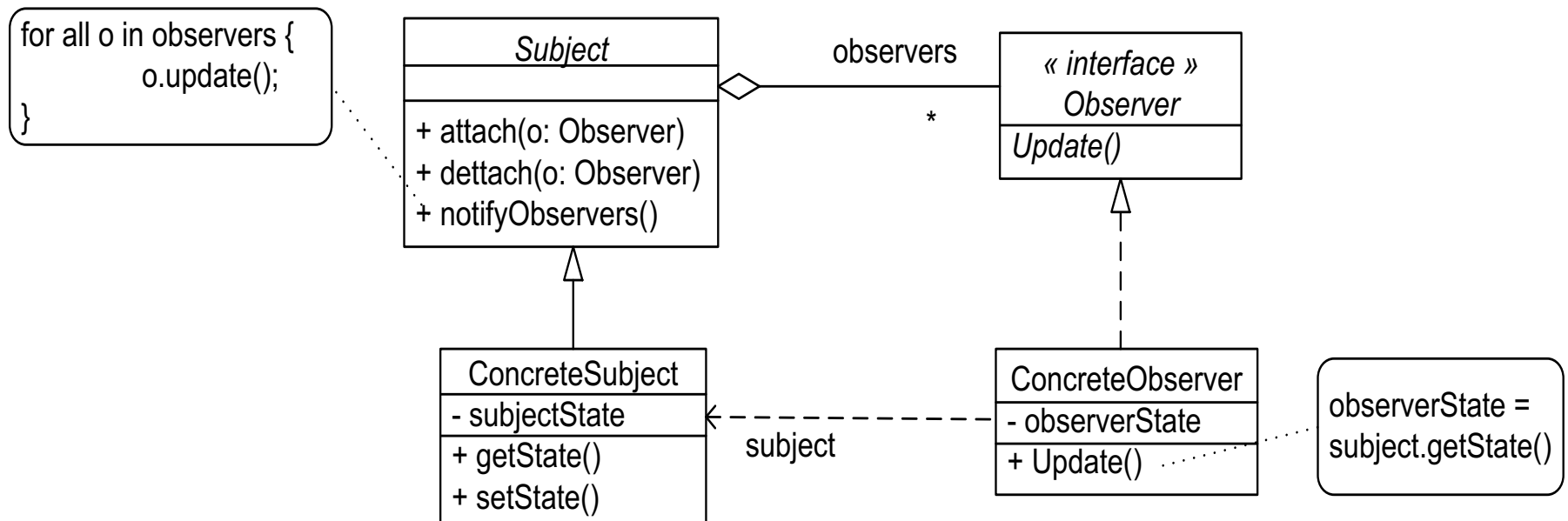
□ Solution

- ▣ 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

17

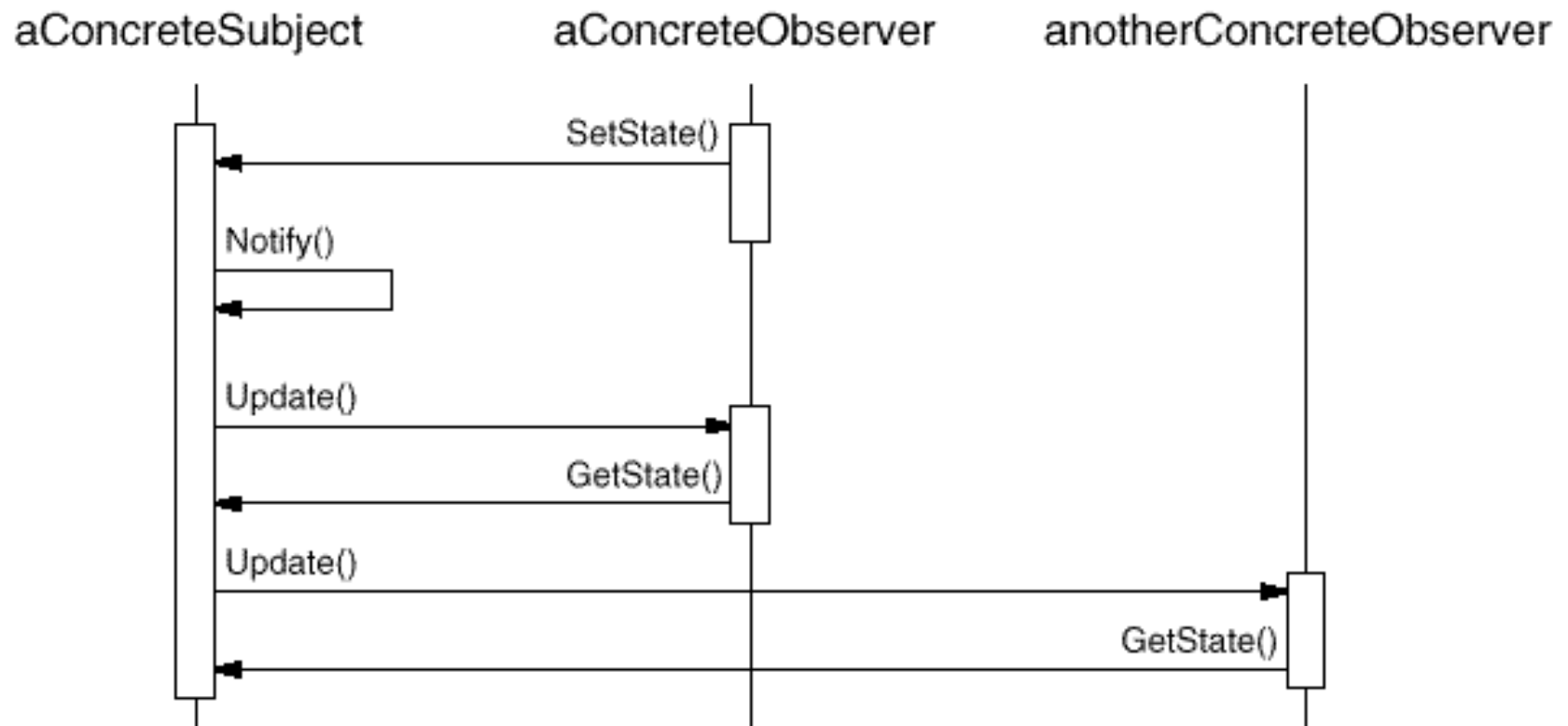
- La structure générique du patron selon GoF
 - C'est cette structure que nous allons utiliser dans le cours



Le patron Observateur

18

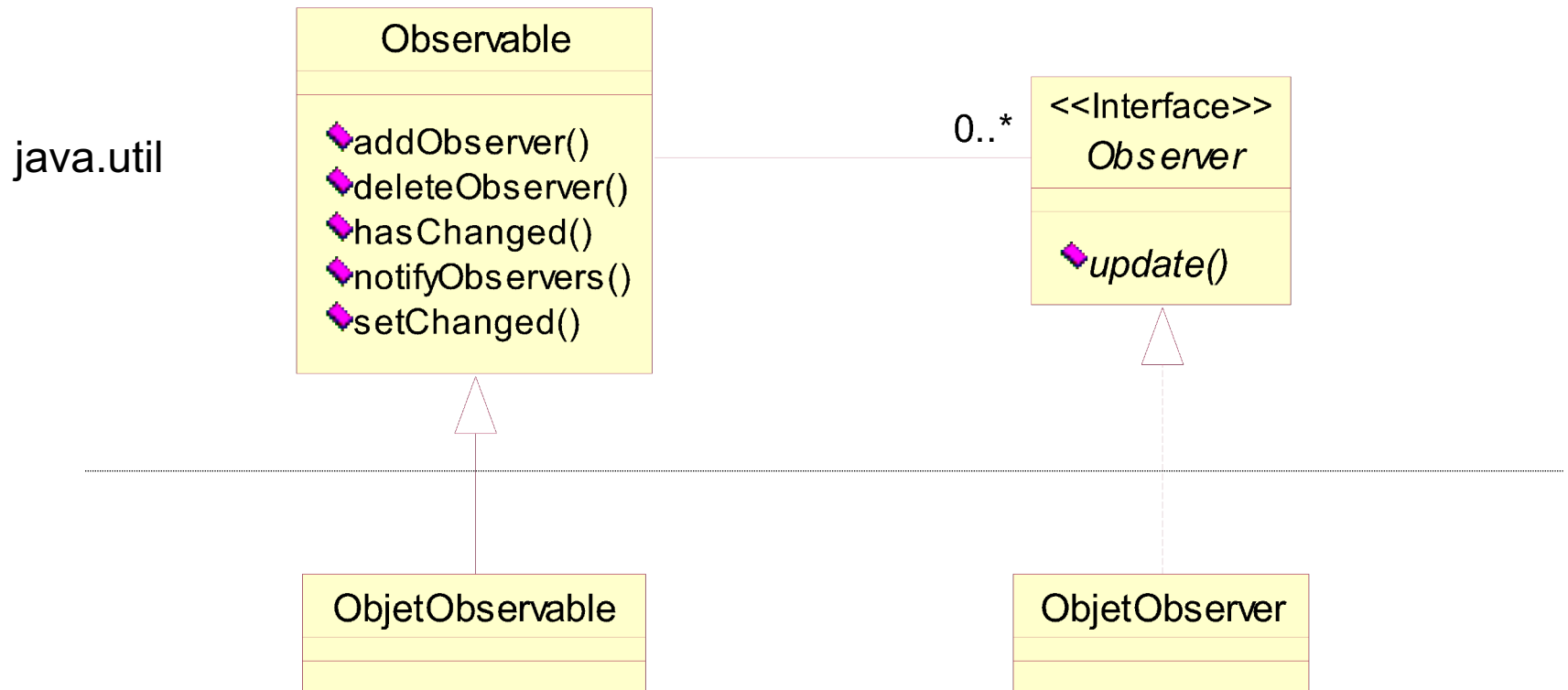
□ Les interactions des entités du patron



Le patron Observateur

19

□ L'observateur dans Java



Le patron Observateur

20

- 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)

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

Exemple de problème de conception

22

```
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

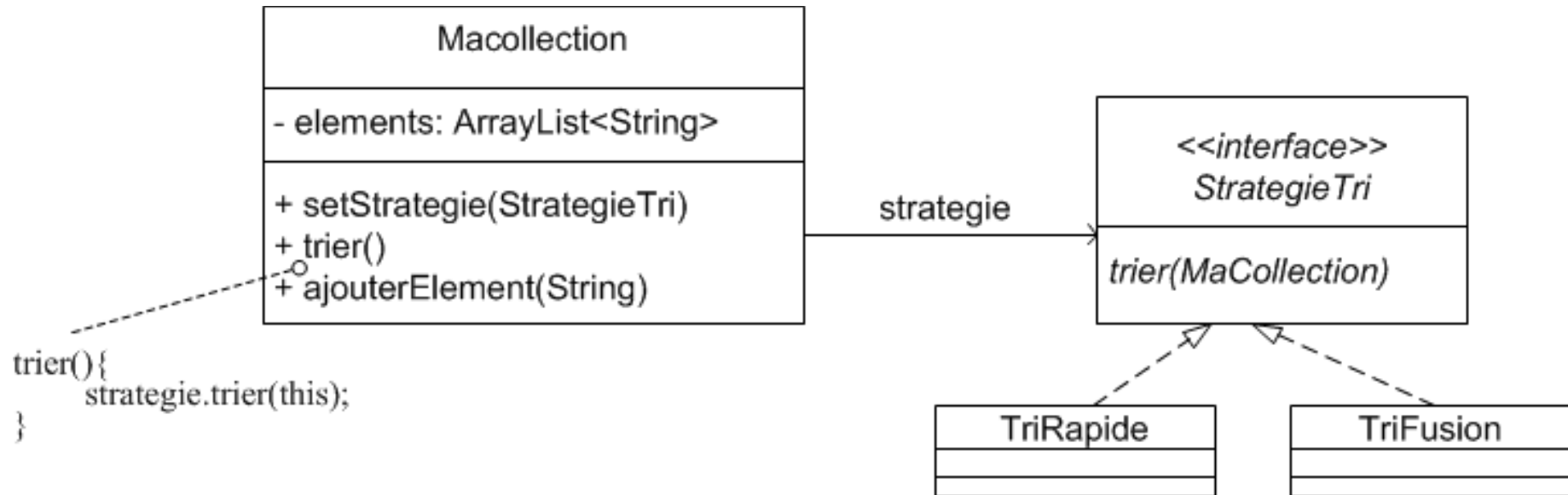
23

- ❑ 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

- ❑ 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

25



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

Deuxième exemple

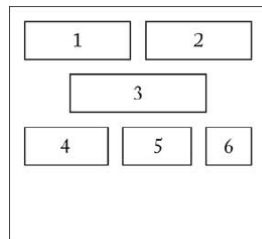
26

- L'interface utilisateur 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

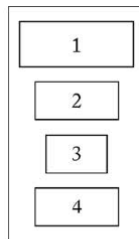
Layout manager (Gestionnaire de disposition)

27

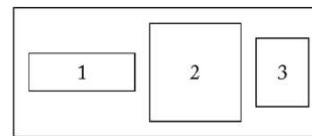
- *FlowLayout*: de gauche à droite, commence une nouvelle rangée lorsque celle courante est pleine
- *BoxLayout*: de gauche à droite ou de haut en bas
- *BorderLayout*: 5 zones, Center, North, South, East, West
- *GridLayout*: grille, tous les composants ont la même taille
- *GridBagLayout*: complexe,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



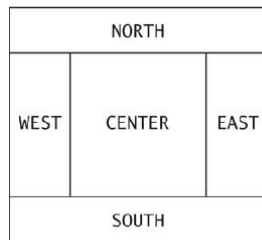
FlowLayout



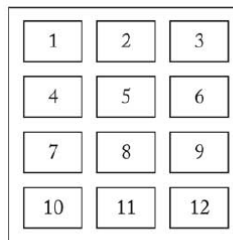
BoxLayout (vertical)



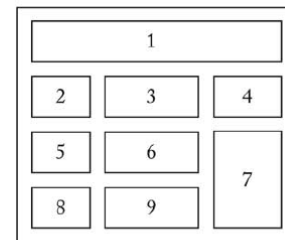
BoxLayout (horizontal)



BorderLayout



GridLayout



GridBagLayout

Layout manager (Gestionnaire de disposition)

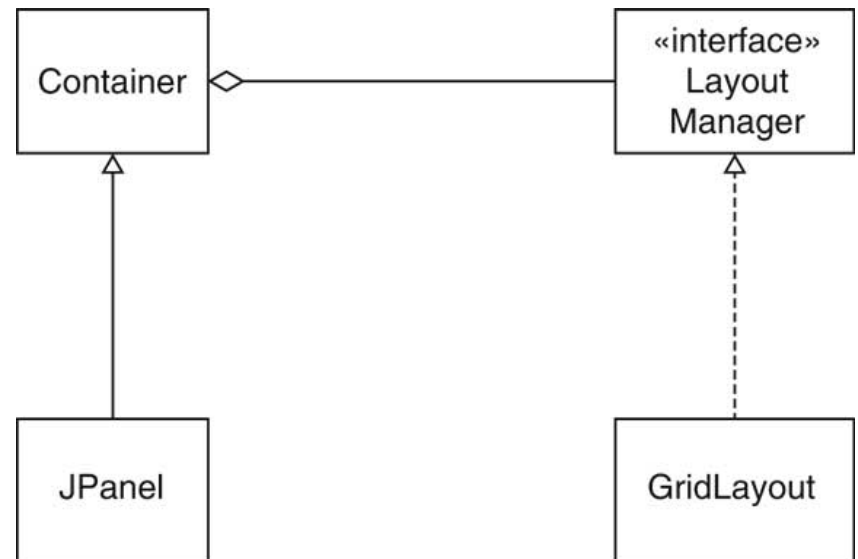
28

□ 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

29

- 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 text area for the speaker's message, a numeric keypad, and a text area for the microphone's message. At the bottom, there 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)

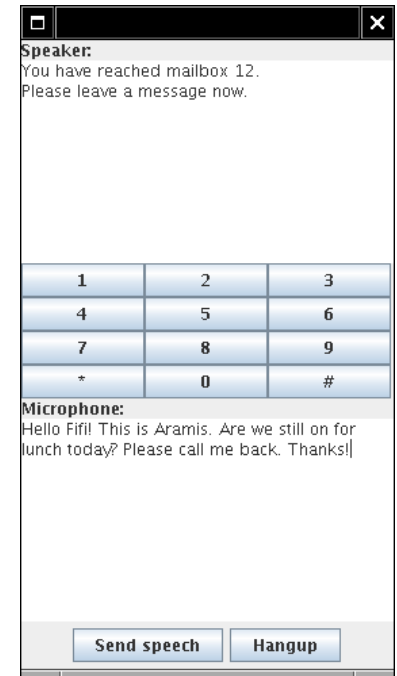
30

□ 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);
```

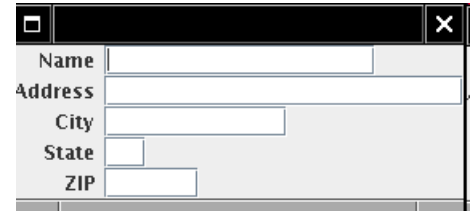


Gestionnaire personnalisé de disposition

31

- 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

32

- 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

Le patron Stratégie

34

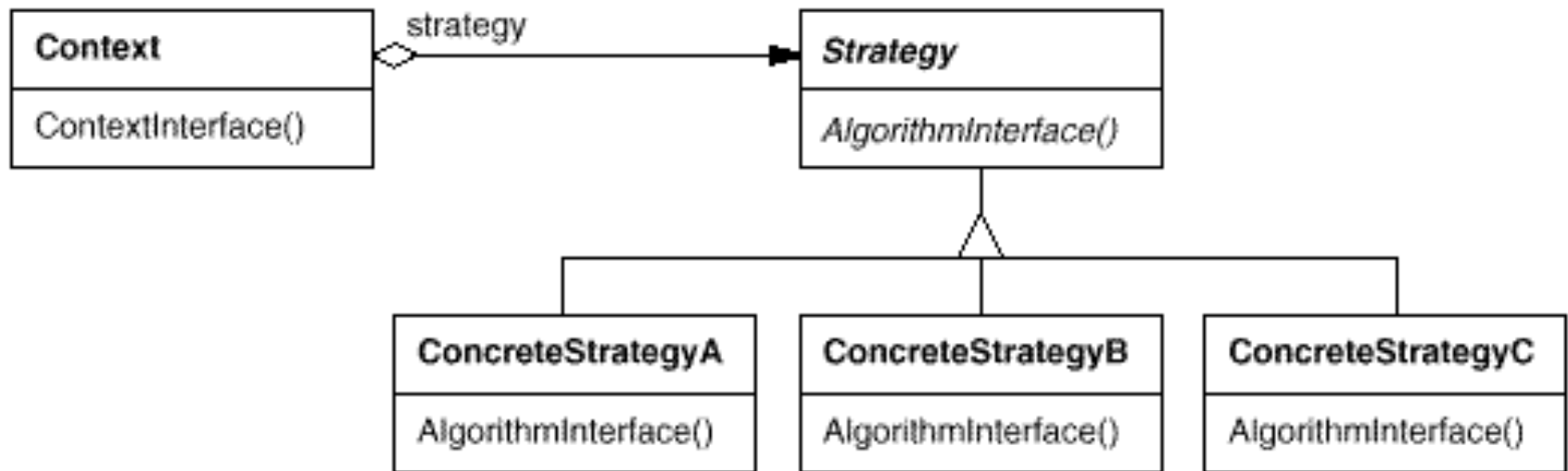
□ Solution

- ▣ 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

35

□ La structure générique du patron selon GoF



Nom dans le patron de conception	Nom dans l'exemple de tri des collections
Context	Collections
Strategy	Comparator
ConcreteStrategyA	une classe implémentant Comparator
AlgorithmInterface()	compare ()

Le patron Stratégie

36

- Quels sont les avantages et inconvénients du patron Stratégie?

Le patron Stratégie

37

□ 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.