ÉTS
Le génie pour l'industrie

LOG121

Conception orientée objet

Patrons Commande et Singleton

Enseignante: Souad Hadjres

Plan

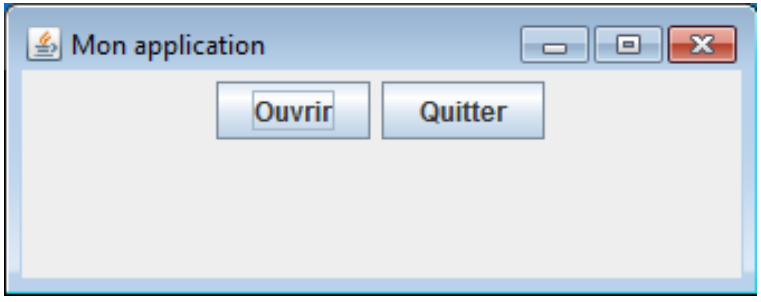
- Patron Commande
- Patron Singleton

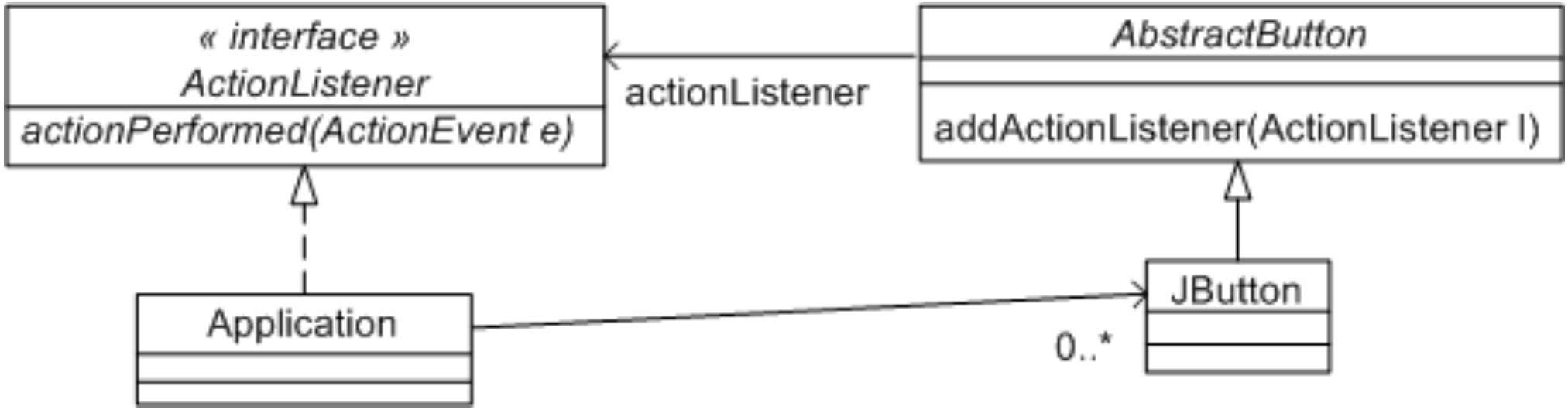
Exemple de problème de conception

- Le framework Swing offre un certain nombre d'objets graphiques tels que des boutons.
- Ces objets graphiques émettent des requêtes lorsqu'un utilisateur les actionne.
- Les concepteurs de Swing ne savaient pas quels objets vont recevoir la requête ni comment elle va être exécutée.
  - Ce sont les applications utilisant Swing qui savent quelle requête est émise et quel objet doit réagir à cette requête.

Exemple de problème de conception

- Un exemple d'application simple





Exemple de problème de conception

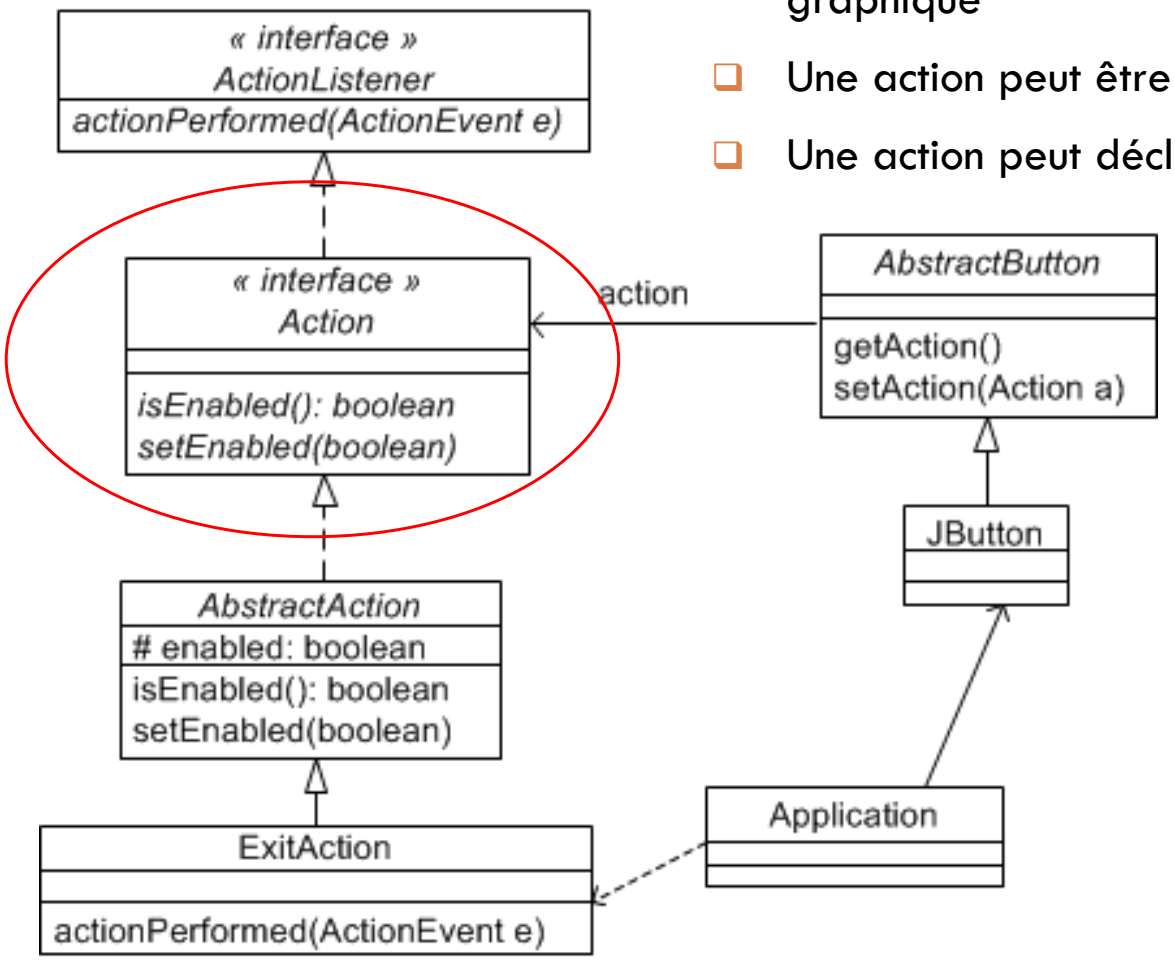

```
public class UneApplication extends JFrame implements ActionListener {
  private JButton openButton = new JButton("Ouvrir");
  private JButton exitButton = new JButton("Quitter");
  public UneApplication() {
     super("Mon application");
     setSize(400, 400);
     JPanel myPanel = new JPanel();
     myPanel.add(openButton);
     myPanel.add(exitButton);
     add(myPanel);
     openButton.addActionListener(this);
     exitButton.addActionListener(this);
  }
  public void actionPerformed(ActionEvent event) {
     if (event.getSource()== openButton)
        JOptionPane.showMessageDialog((Component)event.getSource(), "Ouvrir");
     else if (event.getSource()== exitButton)
        System.exit(0);
  }
  public static void main(String[] argv) {
     UneApplication application = new UneApplication();
     application.setVisible(true);
  }
}
```

Exemple de problème de conception

- La méthode actionPerfomed dans mon application n'est pas très pratique: elle se complexifiera rapidement avec l'ajout d'autres boutons ou composants graphiques.
- Aussi l'application va avoir pas mal d'autres méthodes et beaucoup trop de responsabilités.
- La même action (exit par exemple) peut être exécutée de différentes façons (à partir d'un bouton, un item dans un menu…).
- Une action peut être non permise dans certains contextes
  - par exemple, on ne peut faire « coller » sans avoir fait « copier » ou « couper » auparavant dans un éditeur
- Comment les concepteurs de Swing ont réglé ces problèmes?

Solution au problème

- L'interface Action étend ActionListener
- Une action peut avoir un nom et une icône qui lui est associée
- Une action peut être associée à différents éléments de l'interface graphique
- Une action peut être activée ou désactivée
- Une action peut déclarer une autre action comme opposée



Solution au problème


```
public class ApplicationWithActions extends JFrame {
  private JButton openButton = new JButton("Ouvrir");
  private JButton exitButton = new JButton("Quitter");
  public ApplicationWithActions() {
       super("Mon application");
       setSize(400, 400);
       JPanel myPanel = new JPanel();
       myPanel.add(openButton);
       myPanel.add(exitButton);
       add(myPanel);
       openButton.setAction(new OpenAction());
       exitButton.setAction(new ExitAction());
  }
  public static void main(String[] argv) {
       ApplicationWithActions application = new ApplicationWithActions();
       application.setVisible(true);
  }
}
```

```
public class OpenAction extends  AbstractAction{

    public OpenAction() {
        super("Open");
    }
    public void actionPerformed(ActionEvent e){
        JOptionPane.showMessageDialog((Component)e.getSource(), "Ouvrir");
    }

}
```

```
public class ExitAction extends AbstractAction {
  public ExitAction() {
     super("exit");
  }
  public void actionPerformed(ActionEvent e) {
     System.exit(0);
  }
}
```

Quand on attache une action à un composant à l'aide de la méthode setAction:

-L'état du composant est mis à jour pour correspondre à l'état de l'action.

-L'objet Action est enregistré comme écouteur (actionListener) sur le composant.

-Si l'état de l'action change, l'état du composant est mis à jour pour correspondre à l'action

Solution au problème

- On a encapsulé les actions à exécuter dans des objets qui étendent la classe AbstractAction
- Ces objets s'occupent d'exécuter les actions
- Ils peuvent être utilisés par plusieurs composants graphiques
- On peut aussi leur associer des états (enabled, disabled)
- Cela permet aussi de gérer les actions (par exemple, les composer, les annuler (undo), etc.)
- **On a aussi découplé l'objet source de l'action de celui sur lequel l'action s'exécute**

```
public class CommandTester
{
   public static void main(String[] args)
   {
      JFrame frame = new JFrame();
      JMenuBar bar = new JMenuBar();
      frame.setJMenuBar(bar);
      JMenu menu = new JMenu("Say");
      bar.add(menu);
      JToolBar toolBar = new JToolBar();
      frame.add(toolBar, BorderLayout.NORTH);
      JTextArea textArea = new JTextArea(10, 40);
      frame.add(textArea, BorderLayout.CENTER);

      GreetingAction helloAction = new GreetingAction(
            "Hello, World", textArea);
      helloAction.putValue(Action.NAME, "Hello");
      helloAction.putValue(Action.SMALL_ICON,
         new ImageIcon("hello.png"));

      GreetingAction goodbyeAction = new GreetingAction(
            "Goodbye, World", textArea);
      goodbyeAction.putValue(Action.NAME, "Goodbye");
      goodbyeAction.putValue(Action.SMALL_ICON,
            new ImageIcon("goodbye.png"));

      helloAction.setOpposite(goodbyeAction);
      goodbyeAction.setOpposite(helloAction);
      goodbyeAction.setEnabled(false);

      menu.add(helloAction);
      menu.add(goodbyeAction);

      toolBar.add(helloAction);
      toolBar.add(goodbyeAction);

      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.pack();
      frame.setVisible(true);
   }
}
```

```
public class GreetingAction extends AbstractAction
{
   /**
      Constructs a greeting action.
      @param greeting the string to add to the text area
      @param textArea the text area to which to add the greeting
   */
    private String greeting;
    private JTextArea textArea;
    private Action oppositeAction;

   public GreetingAction(String greeting, JTextArea textArea)
   {
      this.greeting = greeting;
      this.textArea = textArea;
   }

   /**
      Sets the opposite action.
      @param action the action to be enabled after this action was
      carried out
   */
   public void setOpposite(Action action)
   {
      oppositeAction = action;
   }

   public void actionPerformed(ActionEvent event)
   {
      textArea.append(greeting);
      textArea.append("\n");
      if (oppositeAction != null)
      {
         setEnabled(false);
         oppositeAction.setEnabled(true);
      }
   }
}
```

Le patron Commande

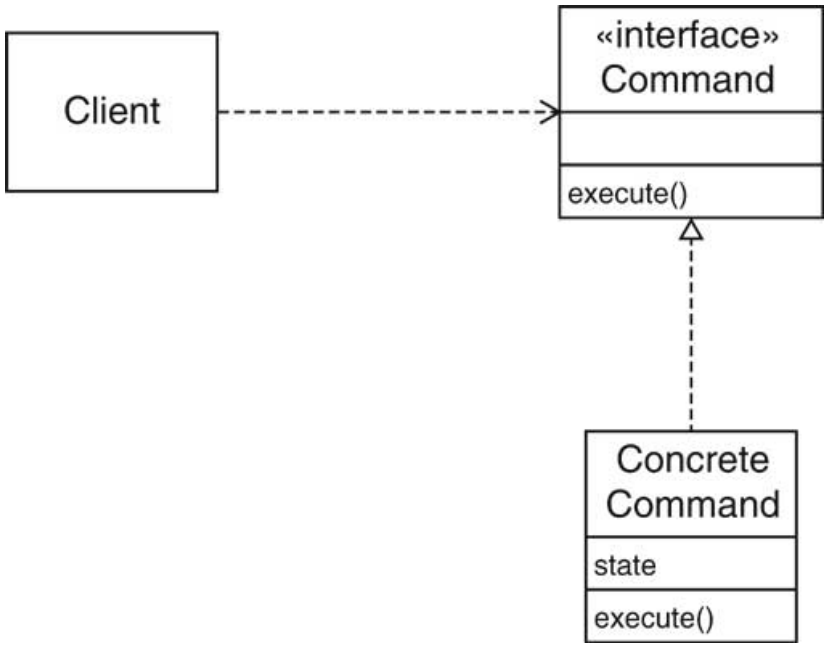
- Contexte
  - On veut implémenter des commandes qui se comportent comme des objets, soit parce qu'on a besoin d'associer des informations aux commandes (états) ou parce qu'on veut garder trace des commandes pour pourvoir les ré-exécuter plusieurs fois (redo), les défaire (undo) ou les regrouper (des macros).
  - On veut découpler l'objet qui invoque la commande de celui qui va la recevoir.

Le patron Commande

- Solution
  - Définir une interface Command avec une méthode pour exécuter la commande.
  - Cette interface définit les méthodes permettant d'accéder à l'état de la commande.
  - Chaque classe concrète qui représente une commande implémente cette interface.
  - Pour invoquer la commande, appeler la méthode d'exécution.

Le patron Commande

- La structure du patron dans Horstman



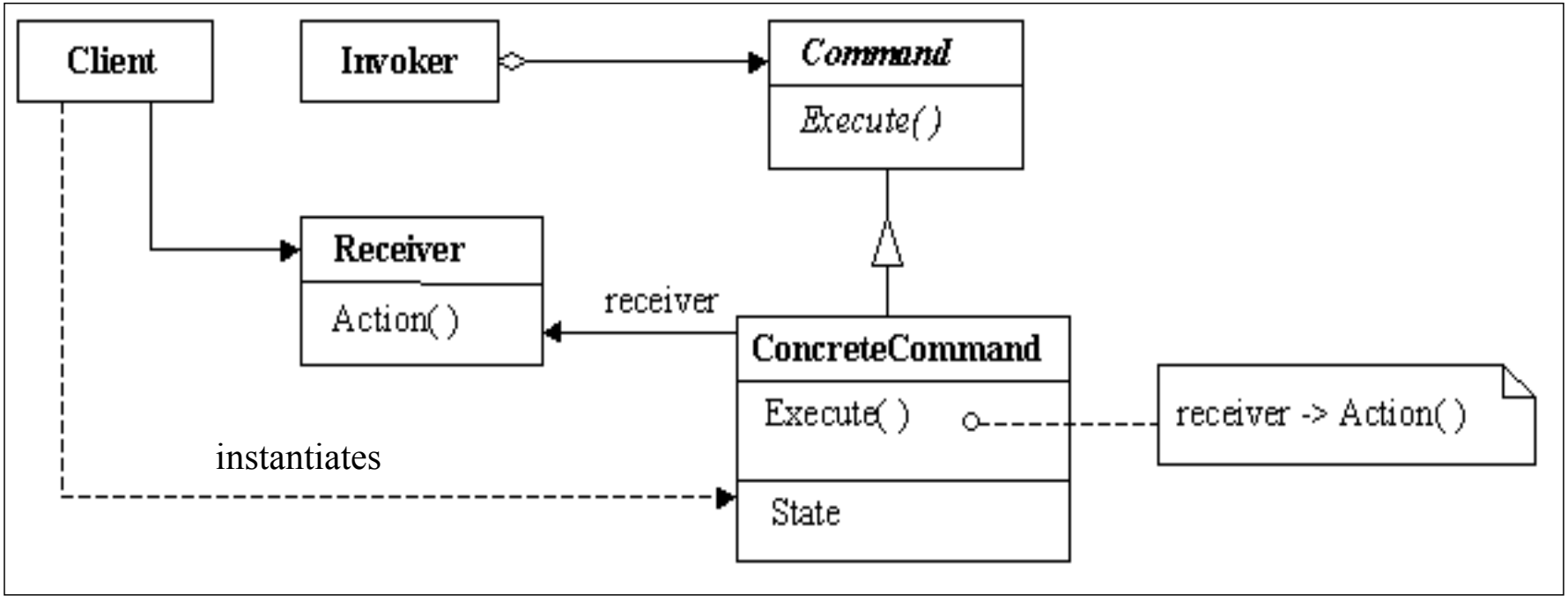
Cette version est très simplifiée:

1- on ne voit pas les objets sur lesquels les commandes concrètes agissent

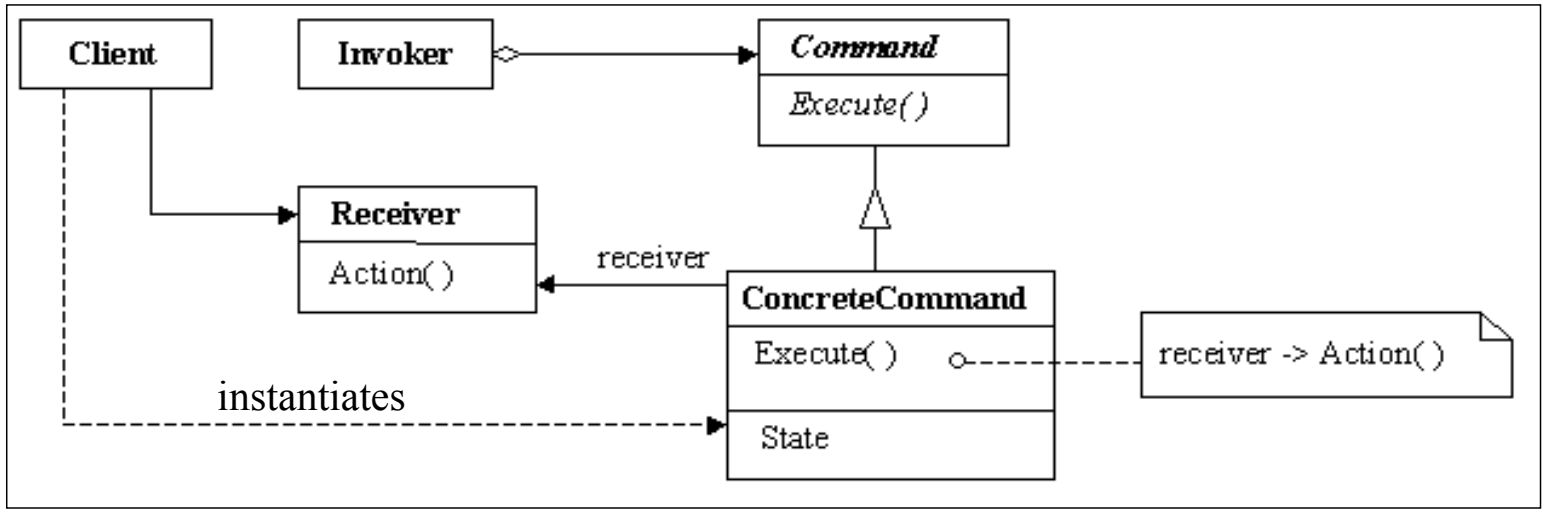
2- On ne voit pas les objets pouvant être à l'origine de l'exécution des commandes concrètes (ex. l'équivalent des composants swing de l'exemple précédent)

Le patron Commande

- La structure générique du patron dans GoF
  - *C'est cette version plus complète que nous utiliserons dans le cours*



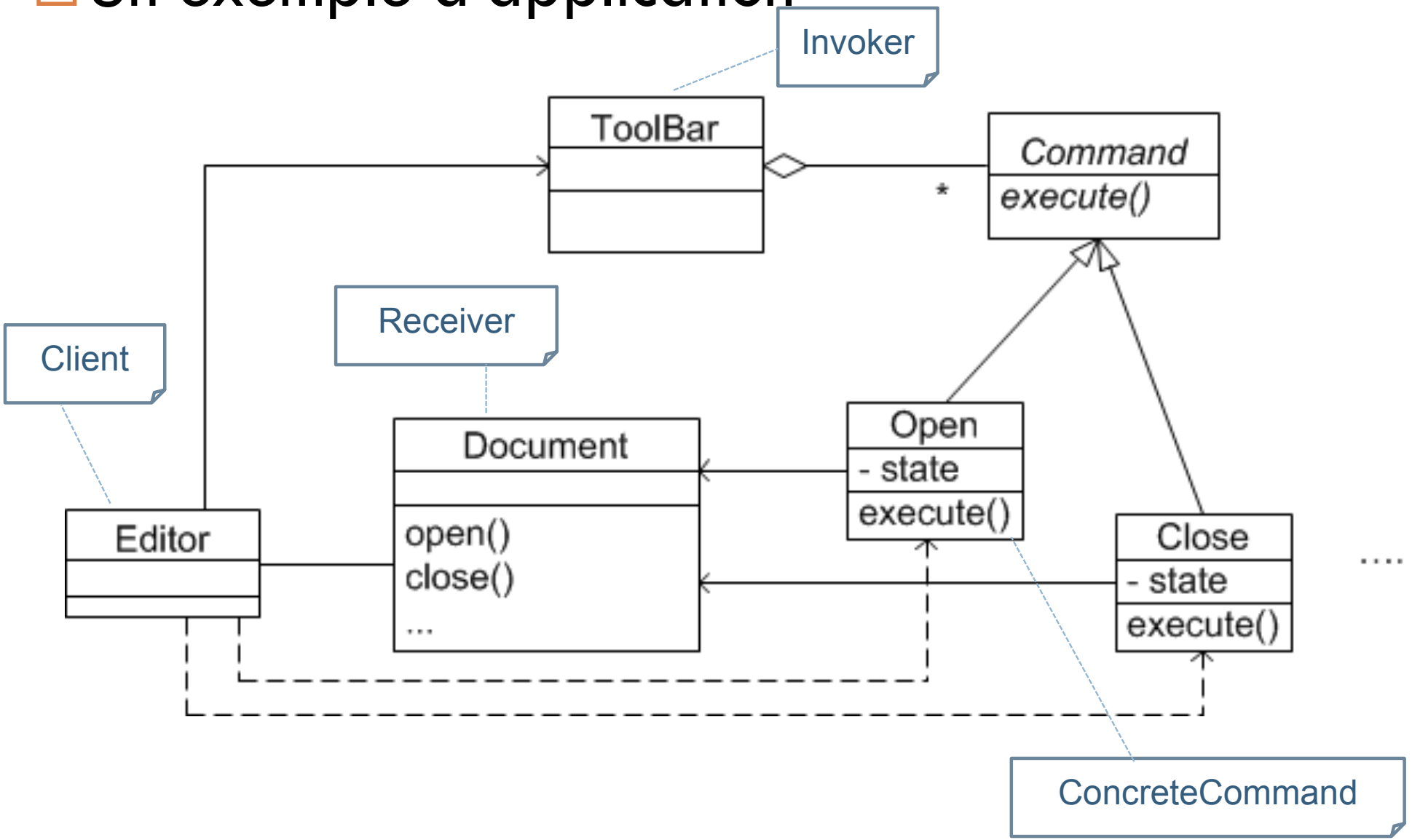
Le patron Commande



Nom dans le patron de conception	Nom dans l'exemple de Swing
Command	Action
ConcreteCommand	ExitAction
execute()	actionPerformed()
state	nom, icône, enabled
Invoker	Jbutton
Receiver	System (dans le cas de ExitAction)
Client	Application

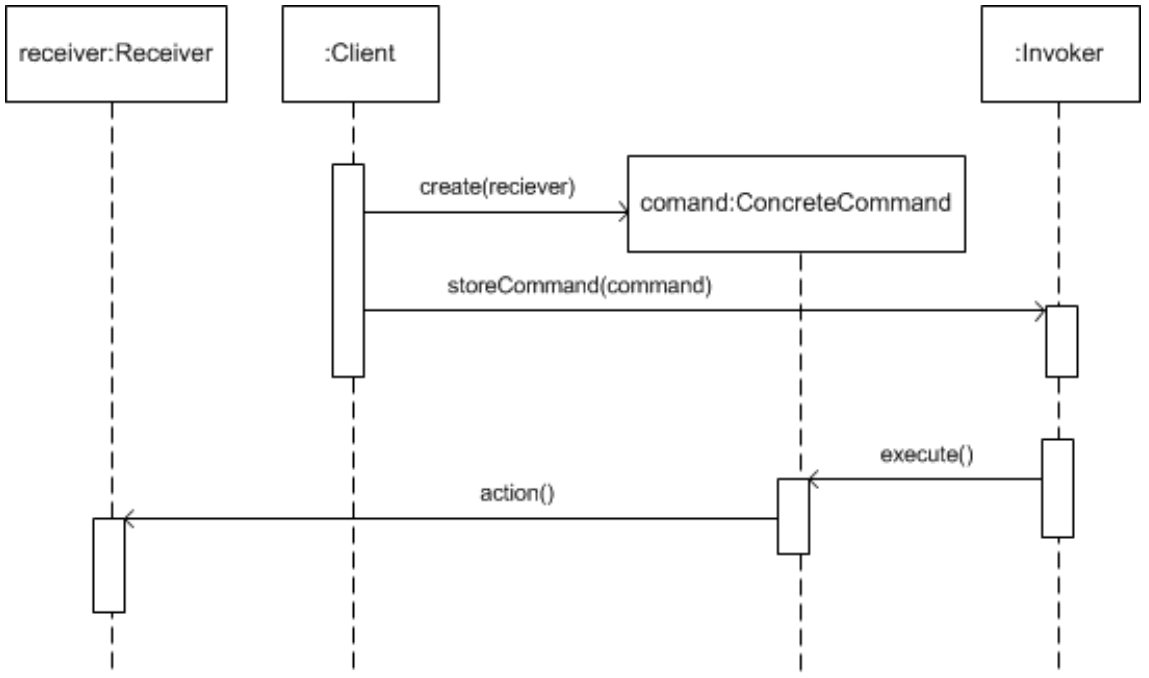
Le patron Commande

- Un exemple d'application



Le patron Commande

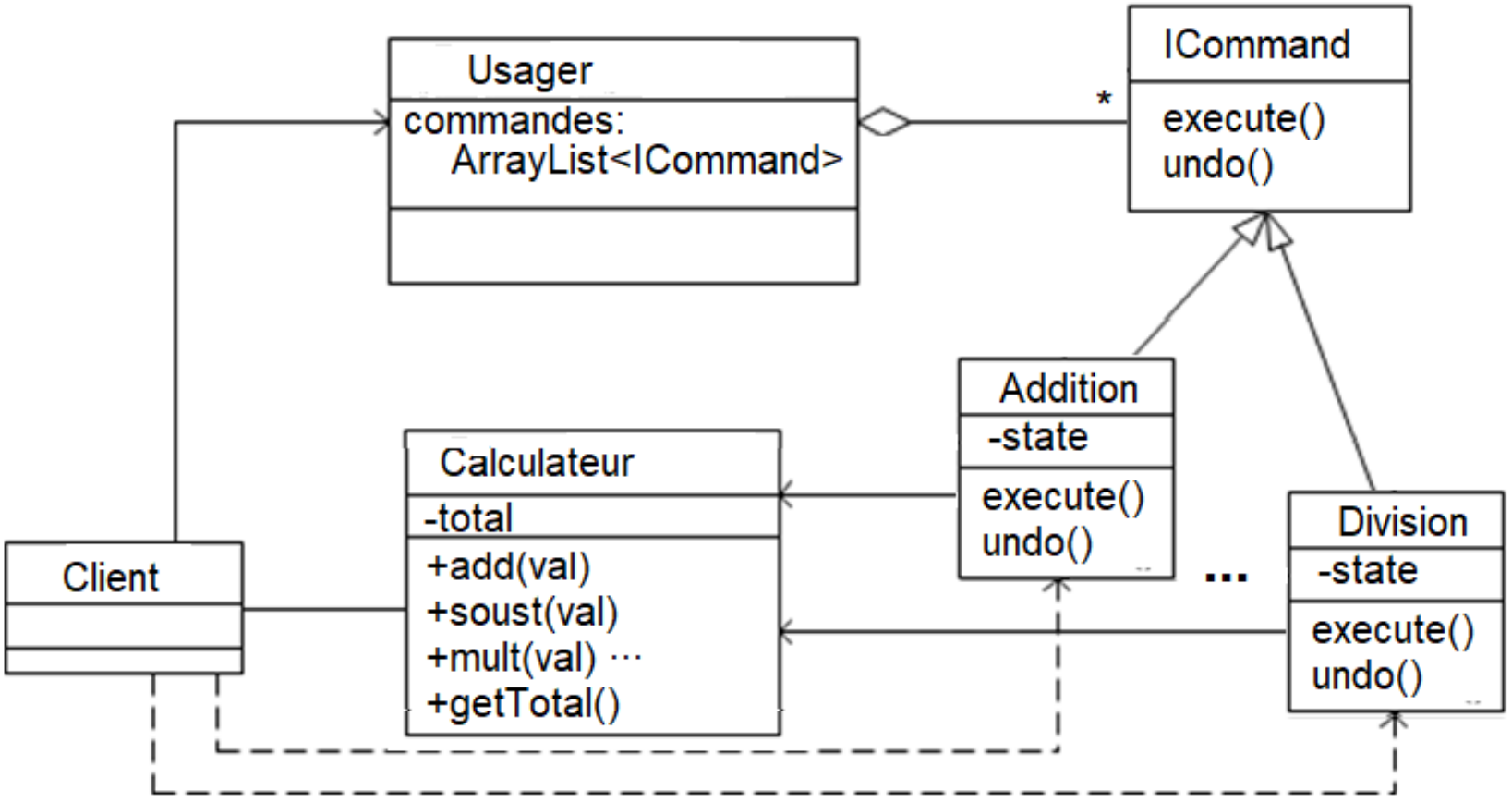
- Les interactions entre les objets du patron



- Le client crée une commande concrète et spécifie son receveur, et il configure un invocateur avec les commandes concrètes
- L'invocateur appelle la méthode Execute d'un objet ConcreteCommand
- La méthode Execute de la commande concrète invoque une ou plusieurs opérations de l'objet receveur pour satisfaire la requête

Le patron Commande

- Un autre exemple d'application: calculatrice de base qui supporte les 4 opérations de base (commandes) et les retours en arrière (undo)



```java
public interface ICommand {
    public void execute();
    public void undo();
}
```

```java
public class Multiplication implements ICommand {
    private double valeur = 0;
    private Calculateur calc;

    public Multiplication(Calculateur calc, double valeur) {
        this.calc = calc;
        this.valeur = valeur;
    }
    public void execute() {
        this.calc.multiplyBy(this.valeur);
    }
    public void undo() {
        this.calc.divideBy(this.valeur);
    }
}
```

```java
public class Calculateur { //Receiver
    private double total;

    public Calculateur() {
        total = 0;
    }

    public void add(double valeur) {
        this.total += valeur;
    }
    public void substract(double valeur) {
        this.total -= valeur;
    }

    public void multiplyBy(double valeur) {
        this.total *= valeur;
    }

    public void divideBy(double valeur) {
        this.total /= valeur;
    }
    public double getTotal() {
        return this.total;
    }
}
```

```java
public class Addition implements ICommand {
    private double valeur = 0;
    private Calculateur calc;

    public Addition(Calculateur calc, double valeur) {
        this.calc = calc;
        this.valeur = valeur;
    }
    public void execute() {
        this.calc.add(this.valeur);
    }
    public void undo() {
        this.calc.substract(this.valeur);
    }
}
```

```java
public class Usager { //Invoker
    private ArrayList<ICommand> commands = new ArrayList<ICommand>();
    private int current = -1;

    public void storeCommand(ICommand cmd) {
        this.commands.add(cmd);
    }

    public void executeCommand() {
        ICommand cmd;
        try {
            cmd = this.commands.get(this.current + 1);
        } catch (IndexOutOfBoundsException ex) {
            cmd = null;
        }
        if (cmd != null) {
            cmd.execute();
            this.current++;
        }
    }
    public void undoCommand() {
        ICommand cmd;
        try {
            cmd = this.commands.get(this.current);
        } catch (IndexOutOfBoundsException ex) {
            cmd = null;
        }
        if (cmd != null) {
            cmd.undo();
            this.current--;
        }
    }
}
```

Le patron Commande


```java
public class Main { //client

    public static void main(String[] args) {

        Calculateur calc = new Calculateur();
        // 0 + 3 = 3
        ICommand cmd1 = new Addition(calc, 3);
        // 3 * 5 = 15
        ICommand cmd2 = new Multiplication(calc, 5);
        // 15 - 5 = 10
        ICommand cmd3 = new Soustraction(calc, 5);
        // 10 / 3 = 3.33
        ICommand cmd4 = new Division(calc, 3);

        Usager usager = new Usager();
        usager.storeCommand(cmd1);
        usager.storeCommand(cmd2);
        usager.storeCommand(cmd3);
        usager.storeCommand(cmd4);

        System.out.println(calc.getTotal());
        usager.executeCommand(); // 0 + 3 = 3
        System.out.println(calc.getTotal());
        usager.executeCommand(); // 3 * 5 = 15
        System.out.println(calc.getTotal());
        usager.executeCommand(); // 15 - 5 = 10
        System.out.println(calc.getTotal());
        usager.executeCommand(); // 10 / 3 = 3.33
        System.out.println(calc.getTotal());
        System.out.println("Annulation des 2 dernières opérations");
        usager.undoCommand(); // 3.33 * 3 = 10
        usager.undoCommand(); // 10 + 5 = 15
        System.out.println(calc.getTotal());
    }
}
```

Les avantages du patron Commande

- Le patron permet de supprimer le couplage entre l'objet qui invoque une commande et celui qui sait comment la réaliser
- L'ensemble des objets « commandes » ont une interface commune qui permet de les invoquer de la même manière
- Les objets « CommandeConcrete » peuvent être assemblés dans une commande composite (ex : transaction)
  - Cela permet de contrôler leur séquencement, les mettre dans des files d'attente et en plus permettre la réversion des opérations (Undo er Redo)

Plan

- Patron Commande
- Patron Singleton

Exemple de problème de conception

- Il existe plusieurs situations où nous avons besoin d'avoir une seule instance d'une classe et d'avoir un accès global à cette instance
- Quelques exemples
  - Une classe représentant un spooler (file d'attente d'une imprimante) ou plus généralement une ressource partagée unique
  - Une classe de fabrication d'objets
  - Une classe qui gère les accès à une base de données

Exemple de problème de conception

- Nous avons un programme où plusieurs classes ont besoin de nombres générés de façon aléatoire
- Un générateur de nombres aléatoires génère, en utilisant des germes, les nombres selon un calcul déterminé!!
  - $Nombre_{i+1} = (a * Nombre_i + b) \% c$
  - $Nombre_0$ est le germe (seed)
- Pour des besoins de déverminage, nous voulons avoir la même séquence de nombres

Solution au problème

- Il faut s'assurer qu'il y a un seul générateur de nombres aléatoires
  - On conçoit une classe dont le constructeur est privé
  - On fournit une méthode statique qui retourne l'instance de cette classe

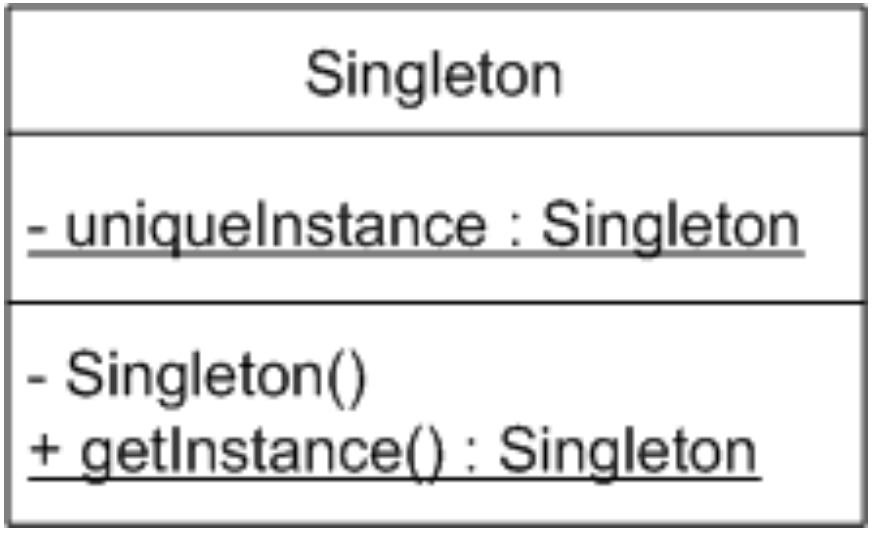
```
public class SingleRandom{

        private Random generator;
        private static SingleRandom instance = new SingleRandom();

        private SingleRandom() {
            generator = new Random();
        }

        public void setSeed(int seed) {
            generator.setSeed(seed);
        }

        public int nextInt() {
            return generator.nextInt();
        }

        public static SingleRandom getInstance(){
            return instance;
        }
}
```

Le patron Singleton

- Contexte
  - Il doit y avoir une instance unique d'une classe.
  - Cette instance doit être accessible.
- Solution
  - Confier à la classe elle-même la responsabilité d'assurer l'unicité de son instance.
    - La classe définit un constructeur privé; Elle construit une seule instance d'elle-même et elle fournit une méthode qui retourne une référence à cette instance unique.
  - Cette classe contrôle comment et quand les classes clientes y accèdent.

Le patron Singleton

- Structure



Le patron Singleton

- Quels sont les avantages de l'application de ce patron?
  - Accès contrôlé à une instance unique;
  - Réduction des variables globales;
  - Permet l'extension de la classe à instance unique;
  - Peut être adapté pour un nombre variable d'instances.

Le patron Singleton

- La classe Math de java définit les méthodes pour réaliser des opérations numériques de base. Elle définit uniquement des méthodes statiques. Est-ce une instance de Singleton?
  - Non, la classe Math ne peut pas être instanciée.
  - Elle ne peut être étendue.
  - C'est une classe qui regroupe un ensemble utile d'opérations numériques (ex. logarithme, fonctions trigonométriques, etc.)

Exemple d'application

- Classe Runtime: est une classe Singleton

```
import java.awt.Desktop;
import java.net.URI;
import java.io.File;

public class Singleton {

    public static void main(String[] args) {
        Runtime run = Runtime.getRuntime();
        System.out.println(run.availableProcessors());
        System.out.println(run.freeMemory());

        Desktop desk = Desktop.getDesktop();
        try {
            desk.browse(new URI("http://google.com"));
            desk.edit(new File("Singleton.java"));
        }catch (Exception e) {System.out.println(e);}

    }
}
```