



Le génie pour l'industrie

LOG121

Conception orientée objet

Groupe 3
Hiver 2020

Patrons Commande et Singleton

Chargé de cours: Cédric St-Onge

Plan

2

- ☐ Patron Commande

- ☐ Patron Singleton

Exemple de problème de conception

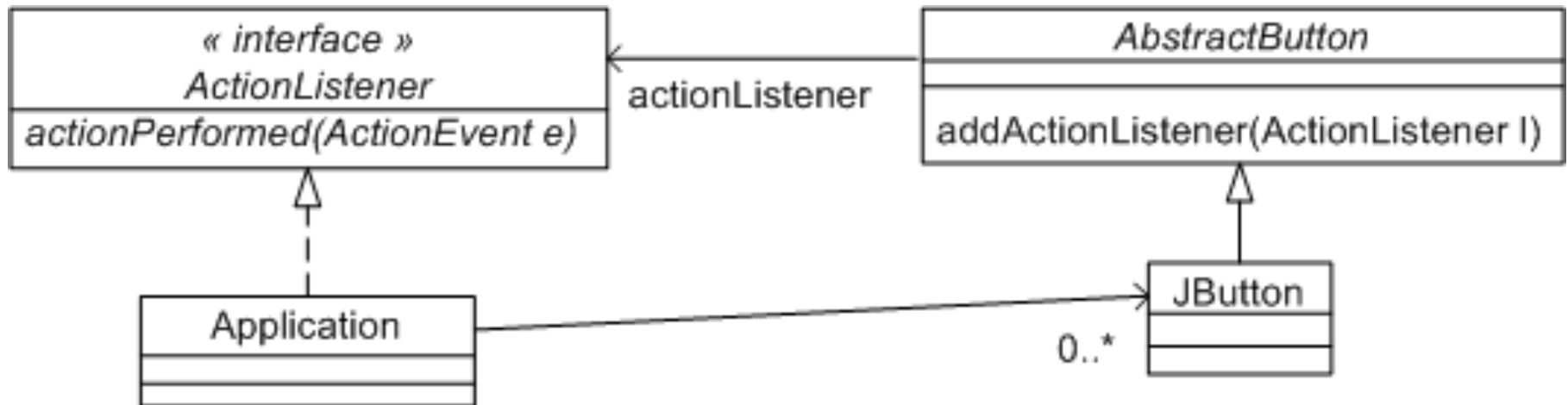
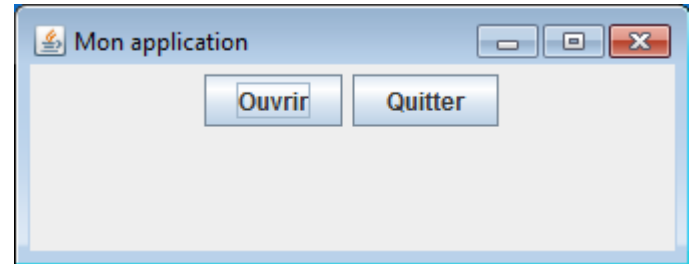
3

- 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

4

□ Un exemple d'application simple



```

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);
    }
}

```

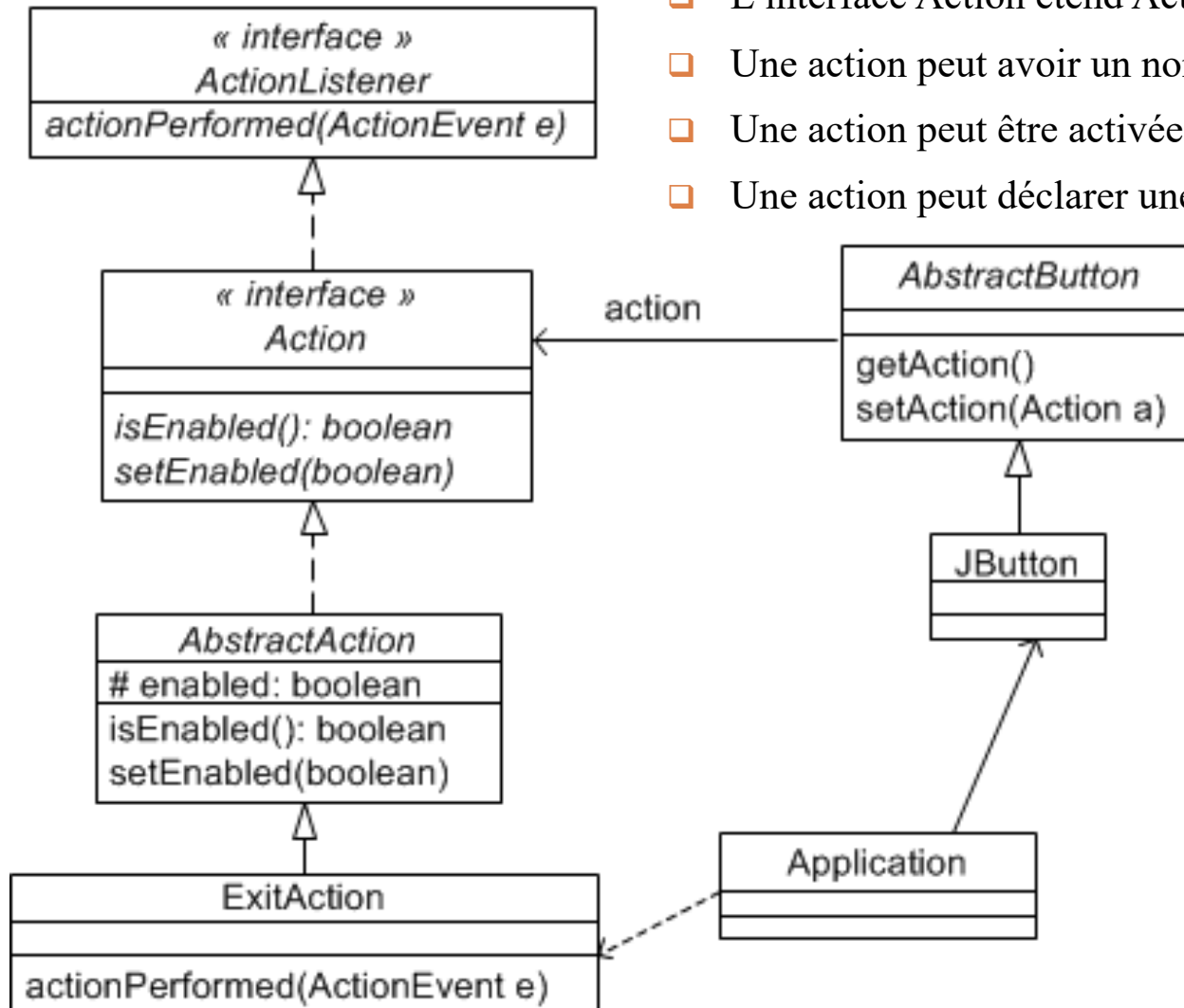
Exemple de problème de conception

6

- ❑ La méthode `actionPefromed` 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).
- ❑ Comment les concepteurs de Swing ont réglé ces problèmes?

Solution au problème

7



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

Solution au p

8

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


Solution au problème

9

- 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**

□ 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 pouvoir 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.

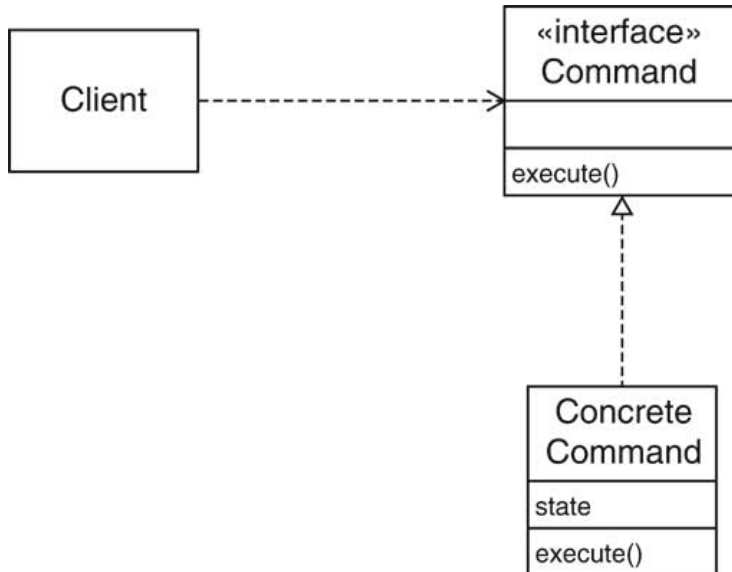
□ Solution

- ▣ 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

12

□ 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)

Nom dans le patron de conception

Command
ConcreteCommand
execute()
state

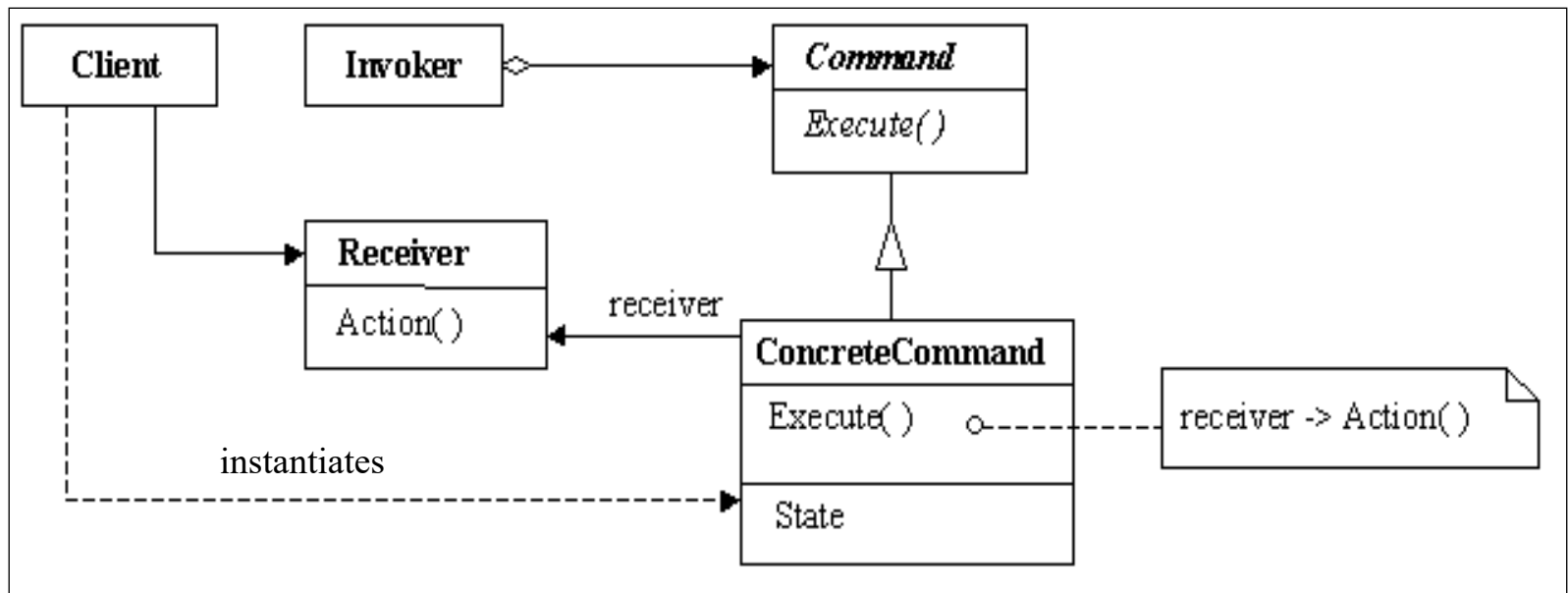
Nom dans l'exemple de Swing

Action
une sous-classe de AbstractAction
actionPerformed()
nom, icône, enabled

Le patron Commande

13

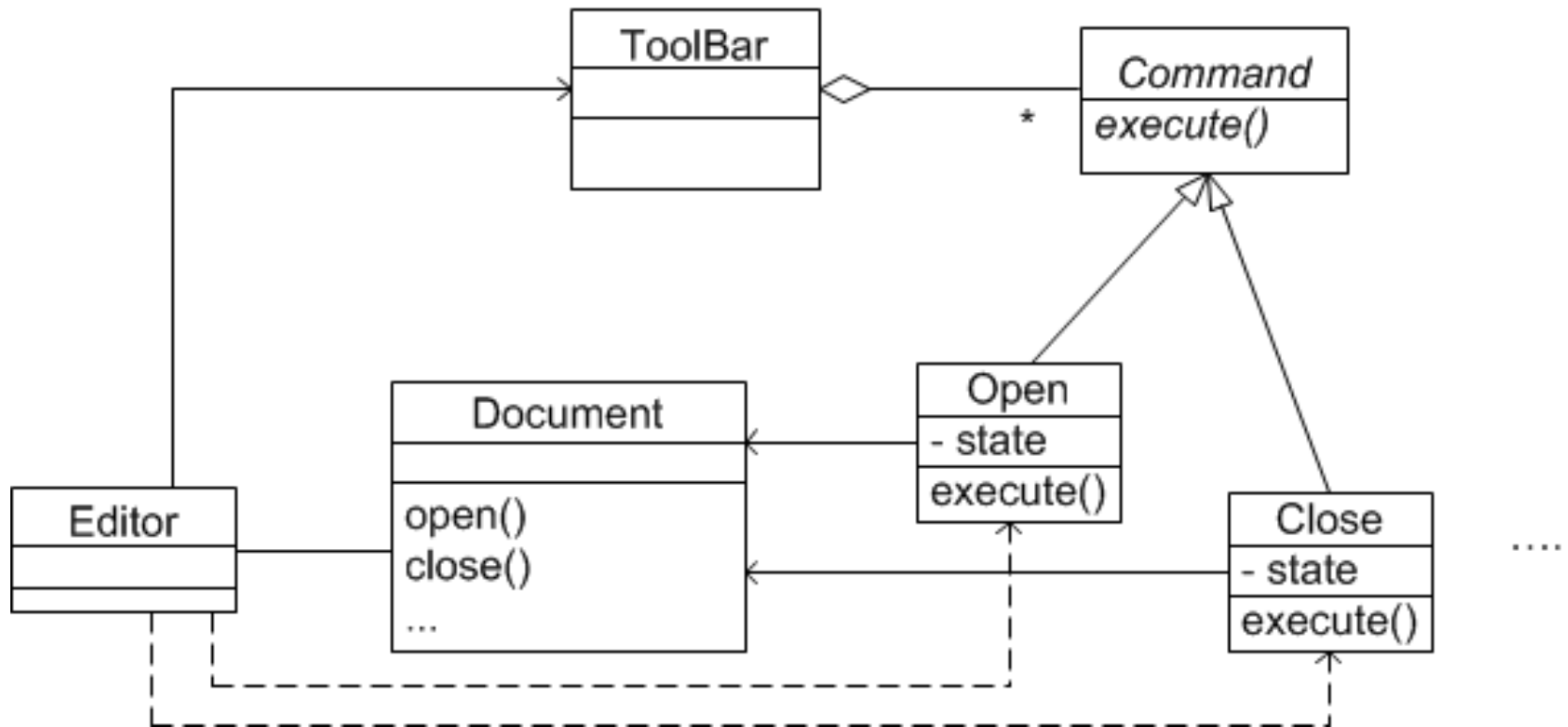
- La structure générique du patron dans GoF
 - *C'est cette version plus complète que nous utiliserons dans le cours*



Le patron Commande

14

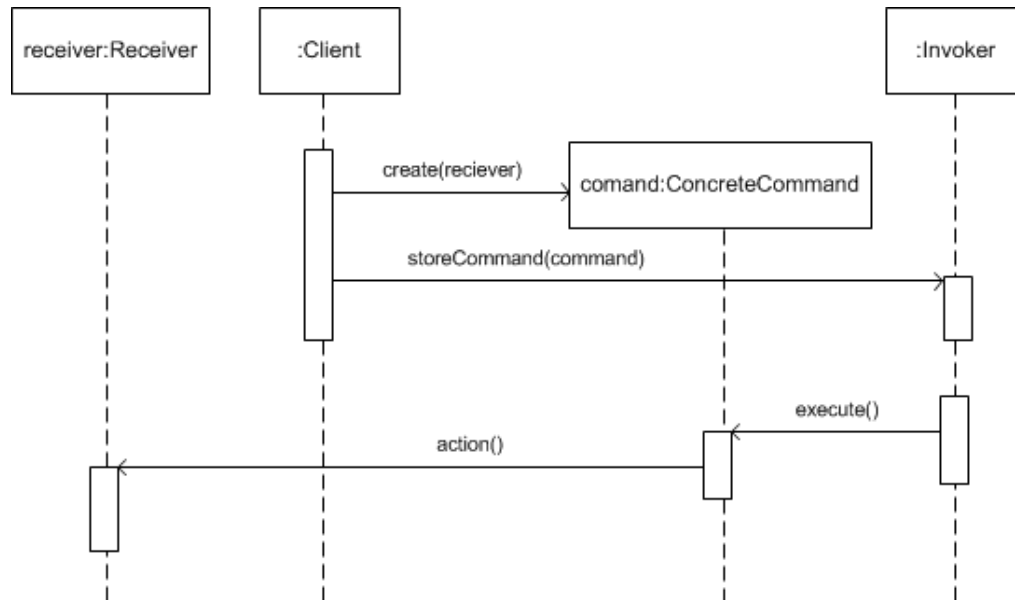
□ Un exemple d'application



Le patron Commande

15

□ 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

Les avantages du patron Commande

16

- 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)

Plan

17

☐ Patron Commande

☐ Patron Singleton

Exemple de problème de conception

18

- 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 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

19

- 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!!
 - ▣ $\text{Nombre}_{i+1} = (a * \text{Nombre}_i + b) \% c$
 - ▣ Nombre_0 est le germe (seed)
- Pour des besoins de déverminage, nous voulons avoir la même séquence de nombres

Solution au problème

20

- 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

21

□ 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

22

□ Structure

Singleton
<u>- uniqueInstance : Singleton</u>
- Singleton() <u>+ getInstance() : Singleton</u>

Le patron Singleton

23

- 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

24

- 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.)