Patterns State, Pattern Command

DUT M3105 : Conception et programmation objet avancées

Serge Rosmorduc
serge.rosmorduc@lecnam.net
Conservatoire National des Arts et Métiers

2018-2019

Pattern état (state)

Motivation

Gérer un objet dont le comportement change de manière considérable selon les moments.
Ce comportement est lié à des *états* de l'objet.

Exemples

- une connexion réseau (état ouvert, fermé, en attente de réponse...)
- la lecture d'un fichier HTML (en train de lire du texte, en train de lire une balise ouvrante, en train de lire une balise fermante...)
- application graphique : selon l'outil correspondant, le fonctionnement de l'application n'est pas du tout le même.

Dans notre programme de dessin...

Nous avons cette classe monstrueuse (je n'ai pas peur des mots) : `SimpleDrawViewControl.java`

- son code est compliqué ;
- il est difficile de comprendre par exemple ce qui se passe lors d'un déplacement ;
- elle fait trop de choses ;
- ses variables d'instances ne sont pas toujours toutes utilisées ;
- leur sens n'est pas clair ;
- l'ajout de nouvelles manipulation devient de plus en plus complexe.

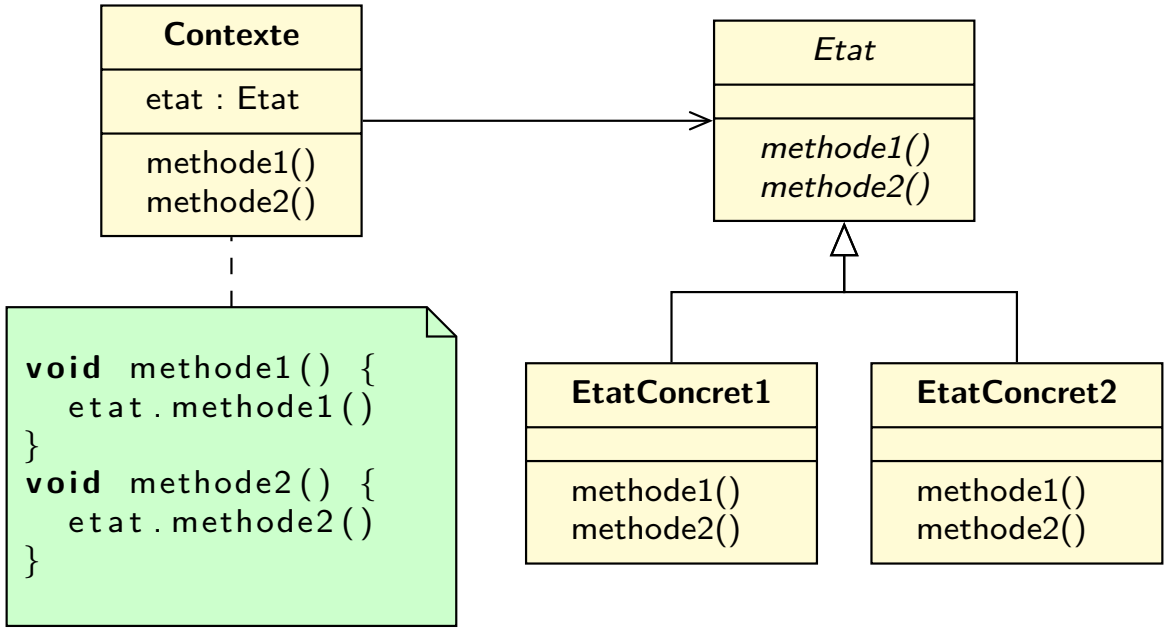
Pour l'instant : les états sont juste des noms (dans l'enum).

Le pattern état

Idée :

- l'application a un état courant ;
- au lieu que les états soient simplement des noms, ce sont de vrais objets ;
- le comportement est géré effectivement par les méthodes des états.

Le pattern état



Dans l'application de dessin

- Le contexte sera représenté par la classe `SimpleDrawViewControl` ;
- les états correspondent à « l'outil » actuellement sélectionné...

```
public class SimpleDrawViewControl
                        extends MouseInputAdapter {
  private Outil outil;
  private SimpleDrawView vue;

  public void setOutil(Outil outil) {
    this.outil= outil;
  }

  public void mouseClicked(MouseEvent e) {
    if (outil != null)
      outil.mouseClicked(e);
  }

  public void mouseDragged(MouseEvent e) {
    if (outil != null)
      outil.mouseDragged(e);
  }
...
}
```

```
public abstract class Outil extends MouseInputAdapter {
  private SimpleDrawView vue;
  protected Dessin getDessin() {
    return vue.getDessin();
  }
  protected abstract void initialiser();

  protected Point getPointFromEvent(MouseEvent e) {
    return vue.coordonneesEcranVersDessin(
                                    e.getX(), e.getY());
  }

  // Notez qu'on a toutes les méthodes de MouseInputAdapter
  // Ce sont elles que nous implémenterons dans
  // les outils particuliers...
}
```

```
public class CreateurCercleOutil extends Outil {
  private Point centre, pointContour;

  public void mousePressed(MouseEvent e) {
    if (centre == null) { // 1er point
      centre = getPointFromEvent(e);
    } else if (pointContour == null) {
      pointContour = getPointFromEvent(e);
      double rayon = Math.sqrt(
                centre.distanceCarre(pointContour));
      if (rayon != 0.0) {
       getDessin().ajouterCercle(centre, rayon);
       initialiser();
      }
    }
  }
  public void initialiser() {
    centre = null;
    pointContour = null;
  }
}
```

Commentaire

- le code du créateur de cercle ne comporte que des informations liées aux cercles ;
- plus rien par exemple sur le déplacement de formes... ;
- on pourrait réutiliser le pattern en interne pour les états des outils (ici, on a deux états, selon qu'on a ou non selectionné le centre) ;
- on multiplie les classes, mais chaque classe est simple et assez lisible ;

Pattern Commande

Motivation

On veut pouvoir manipuler des *actions* : les enregistrer, pour pouvoir éventuellement les rejouer ou les annuler.

Applications :

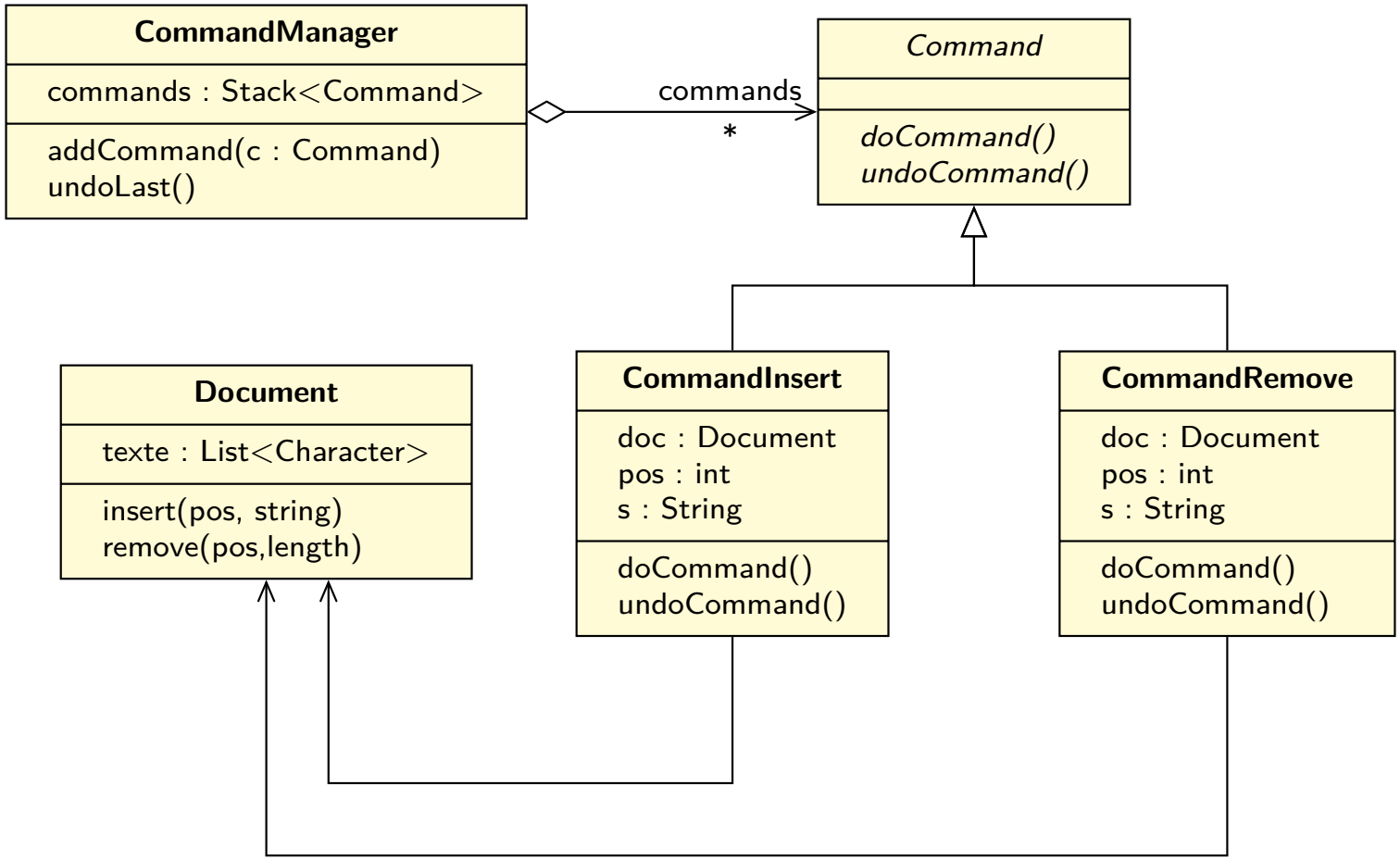
- macros ;
- customization d'une interface graphique ;
- fonction undo/redo

Idée

Une *Commande* représente une **action** et ses **arguments**.

- **Réifier** l'action : la représenter par un objet ;
- la commande contient toutes les informations nécessaires pour s'exécuter.

Implémentation : undo/redo



Implémentation de CommandInsert

```
public class CommandInsert implements Command {
  private Document doc;
  private int pos;
  private String txt;

  public CommandInsert(Document d, int pos, String txt) {
     this.doc= d; this.pos= pos; this.txt= txt;
  }

  public void doCommand() {
    doc.insert(pos, txt);
  }

  public void undoCommand() {
    doc.remove(pos, txt.length());
  }
}
```

Implémentation de CommandRemove

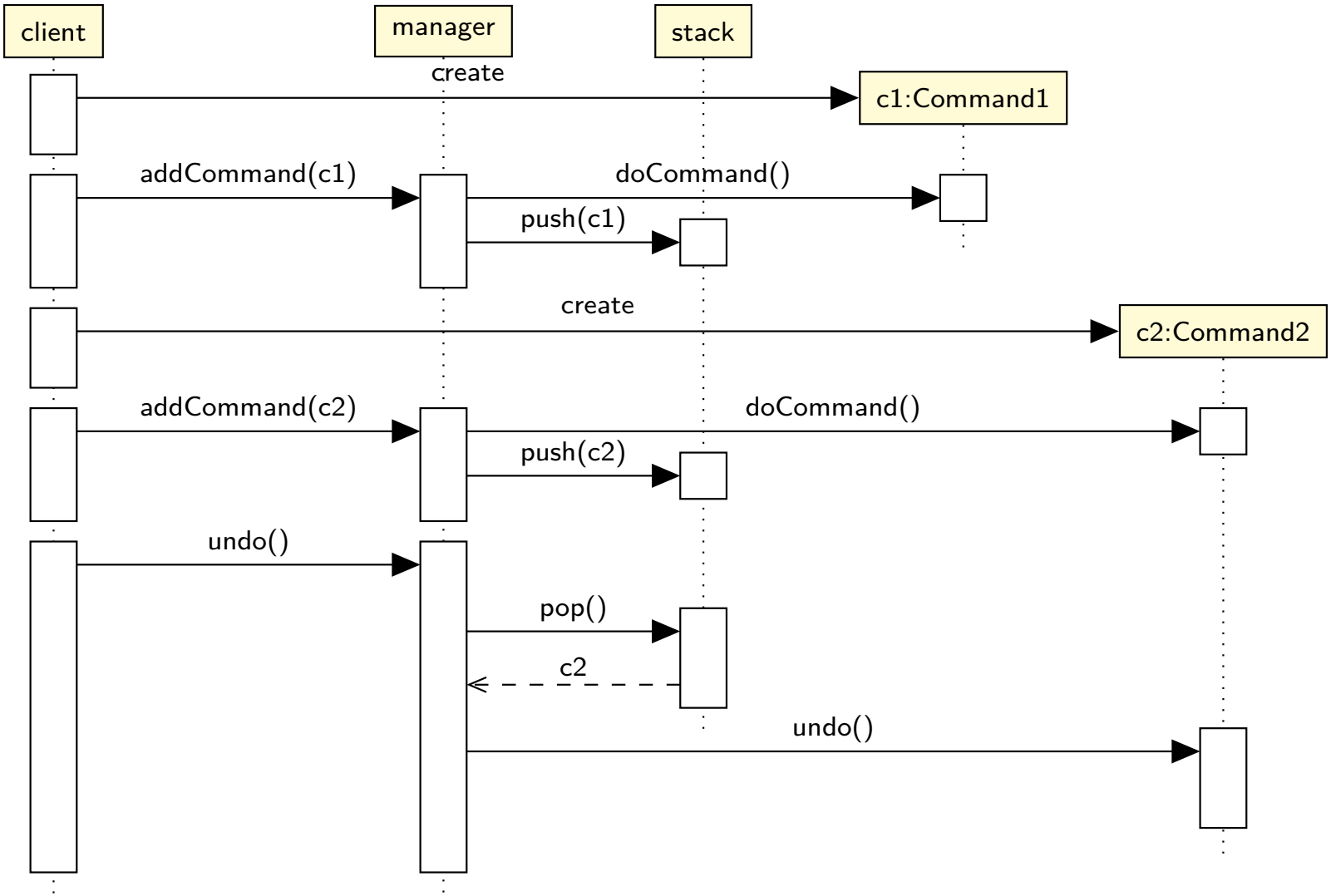
```
public class CommandRemove implements Command {
private Document doc;
private int pos;
private String txt;

public CommandInsert(Document d, int pos, int length) {
    this.doc= d; this.pos= pos;
    this.txt= d.getText(pos, pos+length);
}

public void doCommand() {
  doc.remove(pos, txt.length());
}

public void undoCommand() {
  doc.insert(pos, txt);
}
}
```

Fonctionnement



Variantes du pattern Commande

- pour représenter une commande dans l'absolu (pas un appel particulier avec des arguments spécifiques) ;
- utile pour assigner un comportement particulier à un bouton, un menu, etc... ;
- un actionListener de ce point de vue est une commande ;
- dans notre application, nous utilisons la classe `AbstractAction`, qui est une variante du pattern commande ;
- elle permet d'associer un label, des raccourcis claviers, une icône... à un bouton et/ou à un menu.