Design Patterns Vol. 3

The Path of the Code Samurai

Devert Alexandre
School of Software Engineering of USTC

Table of Contents

undo feature

Let's say you created a program to draw pictures. The user

- starts with an empty drawing
- can place shapes
- can group shapes
- can modify shapes (positions, color, ...)

undo feature

How to allow a user to *undo* some steps of his/her work ?

Macro recording

Let's say you created some data entry application

- user often perform the same sequences of actions
- those sequences are not known in advance
- the sequences might change between users

Macro recording

How to allow a user to store his/her own sequences of actions ?

Multiple user interfaces

One software with same functions, but several user interfaces

- Regular version is for *Windows* PCs
- *iPhone* version should fit in small screen
- Elbonian customers want a different user interface

Multiple user interfaces

How to code the 3 user interfaces, without recoding all the software ?

The Command pattern

For the picture drawing example, we can define several actions

1. Add a shape
2. Group shapes
3. Move a shape
4. Rotate a shape
5. ...

Command interface

Each action is a *command*, with the following interface

```
interface Command {
  void execute(Context context);
}
```

Command interface

A *context* is whatever the actions are able to change

```
class Context {
  ...
}
```

Command interface

For instance, a *context* for drawing might look like this

```
class DrawingContext {
  void addShape(Shape shape);

  void removeShape(Shape shape);

  ...
}


class Shape {
  void rotate(double angle);

  void scale(double ratio);

  void translate(double x, double y);
}
```

Command interface

A *context* for drawing might look like this

```
class DrawingContext {
  void addShape(Shape shape);

  void removeShape(Shape shape);

  ...
}


class Shape {
  void rotate(double angle);

  void scale(double ratio);

  void translate(double x, double y);
}
```

Command interface

Each action implements the *command* interface

```
class AddBox implements Command {
  double x, y, w, h;
  LineStyle lineStyle;
  FillStyle fillStyle;

  void execute(DrawingContext context) {
    shape = new Box(x, y, x + w, y + h);
    shape.setLineStyle(lineStyle);
    shape.setFillStyle(fillStyle);
    context.addShape(shape);
  }
}
```

Command interface

Each action implements the *command* interface

```
class AddCircle implements Command {
  double x, y, radius;
  LineStyle lineStyle;
  FillStyle fillStyle;

  void execute(Context context) {
    shape = new Circle(x, y, radius);
    shape.setLineStyle(lineStyle);
    shape.setFillStyle(fillStyle);
    context.addShape(shape);
  }
}
```

Command interface

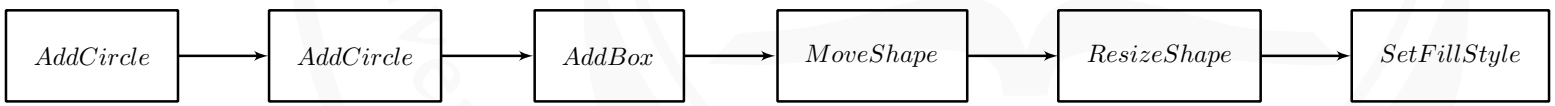
Each action implements the *command* interface

```
class ResizeShape implements Command {
  Shape shape;
  double ratio;

  void execute(DrawingContext context) {
    shape.scale(ratio);
  }
}
```

The Command pattern

The user interface code will

1. Create emphCommand instances
2. Store it in the command history list
3. Execute the command



The Command pattern

A *CommandManager* to handle the command list

```
class CommandManager {
  DrawingContext context;
  List<Command> commandList;

  void addCommand(Command command) {
    commandList.append(command);
    executeCommands();
  }

  void popLastCommand() {
    commandList.pop();
    executeCommands();
  }

  void executeCommands() {
    context.init();
    for(Command command : commandList)
      command.execute(context);
  }
}
```

The Command pattern

Example of command creation, storage and execution

```
...

if (button == addCircleButton) {
  commandManager.addCommand(new AddCircle(x, y, radius, lineStyle, fillStyle));
}

...
```

undo feature

The *undo* feature is now simple to do

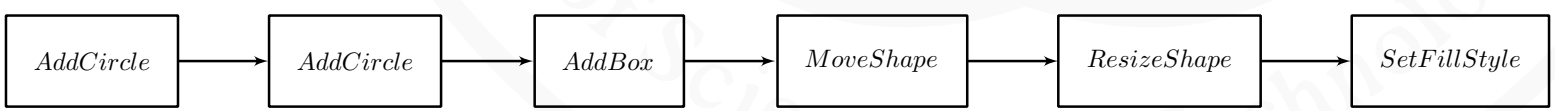
1. Remove the last command performed
2. Execute all the commands from the history

If the commands are expensive, you can store some intermediary steps

undo feature

Example of *undo* and *redo*

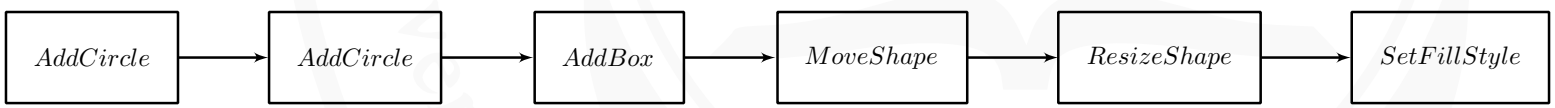
```
...

if (button == undoButton) {
  command = commandManager.getLastCommand();
  commandManager.popLastCommand();
  redoStack.append(command);
}

if (button == redoButton) {
  if (redoStack.size() > 0) {
    commandManager.addCommand(redoStack.top());
    redoStack.pop()
  }
}
...
```



Macro recording

Macro recording is also simple

1. Store the current position of the command history
2. Let the user does his/her job
3. Keep the list of commands $\Rightarrow$ the macro



Multiple user interface

Thanks to the usage of the *Command* pattern

- Different interfaces
- Same set of *Command* implementations

The Command pattern

Actions as objects allows

- Simple history manipulation
- Read/write complex action sequences
- Can help to separate UI and logic

Table of Contents

Event handling

Sometimes, some objects need to know what happens to other objects

Event handling

With an email client $\Rightarrow$ when a contact leaves, the contact list should be updated

Event handling

In a file tree view $\Rightarrow$ when a file is created, the view should be updated

Event handling