Design Patterns and Restructuring

Command

Oliver Haase

HTWG Konstanz
Description

- **Classification**: Object-based behavioral pattern
- **Purpose**: Encapsulate a command as an object. Allows to dynamically configure an invoker with a command object. Invoker can invoke command without knowing its specifics nor the receiver of the command.
- **Also known as**: Action, Transaction
Motivation & Remarks

- Most widely known application of the command pattern: event listeners for GUI programming

Remark: Command pattern is the object-oriented counterpart of function pointers in procedural languages

Remark II: If the command object contains only one operation, function pointers can be considered more lightweight and thus more appropriate
Remark III: In C#, a function pointer is called a delegate. Example:

```csharp
public delegate boolean FUNC(object x, object y);
public boolean Compare(object x, object y) {
    ... } 
public boolean contains(object x, object[] field) {
    CMP(Func myFuncObj = new FUNC(Compare);
    foreach (object obj in field) {
        if (myFuncObj(obj, x))
            return true;
    }
    return false;
}
```
**Remark III:** In C#, a function pointer is called a *delegate*. Example:

```csharp
public delegate bool FUNC(object x, object y);
public bool Compare(object x, object y) {
    // ...
}
public bool contains(object x, object[] field) {
    CMP_Func myFuncObj = new FUNC(Compare);
    foreach (object obj in field) {
        if (myFuncObj(obj, x))
            return true;
    }
    return false;
}
```

**IDIOM**
In Java Swing, when a JButton is pressed, the button invokes the actionPerformed method of a pre-registered ActionListener command object.

The ActionListener interface contains only the actionPerformed method.

an ActionListener command object can be registered using the JButton's addActionListener method.
Remark IV: In Java, an anonymous class might be a good choice to avoid the fully-fledged declaration of a class for a single method that is called only once:

```java
JButton submitButton = new JButton("submit");
submitButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        // do whatever is needed to submit
    }
});
```
Remark IV: In Java, an anonymous class might be a good choice to avoid the fully-fledged declaration of a class for a single method that is called only once:

```java
JButton submitButton = new JButton("submit");
submitButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        // do whatever is needed to submit
    }
});
```
Sample Structure

MyApplication

JButton

ActionListener

actionPerformed()

MyModel

action()

MyActionListener

actionPerformed()  model.action()
Applicability

Use the command pattern if you want to

- dynamically configure an object with an action;
- you want to support → undo functionality;
- you want to keep a log of the executed actions so they can be redone in case of a crash;
- model a complex transaction;
  → in this case the encapsulation of the transaction in a command object reduces code dependencies.
Participants

- **Command**: declares interface for the command invocation
- **ConcreteCommand**:
  - defines the action to be performed
  - defines the receiver of the action
- **Client**: creates a *ConcreteCommand*-object and passes the receiver into it.
- **Invoker**: invokes the command through the *Command* interface
- **Receiver**: knows how to perform the action
Interactions

- Client creates concrete command object and determines the receiver
- Invoker stores the concrete command object
- Invoker executes a request by invoking the command object’s `execute` operation
- If a command can be undone, the command object stores the receiver’s original state so it can be restored later.
- The concrete command object implements the request by calling operation at the receiver object
Undo and Redo

How to implement chains of undos and redos:

1. Provide an additional undo operation in the Command interface.
2. In each ConcreteCommand class, store the state before execution of the execute operation; this may include:
   - all parameters for the action performed by the receiver
   - all state information of the receiver that might change due to the execution of the action (→ memento pattern)
3. Receiver must allow command object to restore receiver's original state.
4. Client stores all executed commands in a command history → depth of command history determines number of possible undo/redo steps.
5. Command objects are copied before insertion into command history.
Undo and Redo

How to implement chains of undos and redos:

- Provide an additional undo operation in the Command interface;
- In each ConcreteCommand class, store the state before execution of the execute operation; this may include:
  - all parameters for the action performed by the receiver
  - all state information of the receiver that might change due to the execution of the action (→*memento* pattern)
- Receiver must allow command object to restore receiver’s original state
- Client stores all executed commands in a *command history* → depth of command history determines number of possible undo/redo steps
- command objects are copied before insertion into command history
Consequences

- Command pattern decouples the entity that invokes a request (*Invoker*) from the one that knows how to implement it (*Command*).
- Command objects can be manipulated (configured) and extended as any other object.
- Command objects can be composed to build macro objects. For this purpose, the composite pattern can be employed.
- New command objects can easily be created without modifying existing classes.
Related Patterns

- The *composite pattern* can be used to build macro commands (in which case the *MacroCommand* class doesn’t point to a receiver object)
- If a command needs to store the receiver’s state (*undo*), it can use the →*memento pattern*
- If a command object needs to be copied before being inserted into the command history, then it behaves like a *prototype*