



AARHUS UNIVERSITET

Software Engineering and Architecture

Pattern Catalog: Command



Command

- In my word processor system I would like the user to configure what the F1 button does freely
 - Like 'save' or 'open new file' or ?
 - Or perhaps record a macro of key strokes in F1
 - F1 => insert text 'iskagefabrik' at the cursor position
- But how to code this?

```
public void F1Press() {  
    editor.showFileDialogAndOpen();  
or  
    editor.save();  
or  
    some other behavior?  
}
```

A parametric solution?

No. Can only handle those case I have imagined in advance ☹

- ③ *Encapsulate what varies.* I need to handle behavior as objects that can be assigned to keys or buttons, that can be put into macro lists, etc. The obvious responsibility of such a “request object” is to be executable. The next logical step is to require that it can “un-execute” itself in order to support undo.
- ① *Program to an interface.* The request objects must have a common interface to allow them to be exchanged across those user interface elements that must enact them. This interface is the **Command** role that encapsulate the responsibility “execute” (and potentially “undo”).
- ② *Object composition.* Instead of buttons, menu items, key strokes hard coding behavior, they delegate to their assigned command objects.

```
editor.save ();
```

becomes something like

```
Command saveCommand = new SaveCommand(editor);  
saveCommand.execute ();
```

Our Command Interface

- Pretty simple!

```
/** The interface defining the Command role */  
interface Command {  
    public void execute();  
    public void undo();  
}
```

- Any instance of this represents *one method invocation to be made to one object*
 - Example:



ps.addPayment(5)



```
class PSInsert5Cent implements Command {  
    private PayStation ps;  
    public void PSInsert5Cent(PayStation ps) {  
        this.ps = ps;  
    }  
    public void execute() {  
        ps.addPayment(5);  
    }  
}
```



```
/** The interface defining the Command role */
interface Command {
    public void execute();
    public void undo();
}
```

```
/** This class is a simple document, it acts as the
 * Receive role in the Command pattern
 */
class Document {
    private List<String> contents = new ArrayList<String>();
    public void write(String text) {
        contents.add(text);
    }
    public void erase(String text) {
        contents.remove(text);
    }
    public String toString() {
        String result = new String();
        for ( String line : contents ) {
            result += line+"\n";
        }
        return result;
    }
}
```

```
/** A concrete command to write text to the document */
class WriteCommand implements Command {
    private Document doc;
    private String line;
    public WriteCommand(Document doc, String line) {
        this.doc = doc; this.line = line;
    }
    public void execute() {
        doc.write(line);
    }
    public void undo() {
        doc.erase(line);
    }
}
```

```
/** A class representing a function key on keyboard */
class FKey {
    private Command command;
    /** assign a command to the key */
    public void assign(Command command) {
        this.command = command;
    }
    /** "press the key" */
    public void press() {
        command.execute();
    }
}
```



```
csdev@m1:~/proj/frsproject/command-demo$ java CommandDemo
```

```
===== Demonstration of Command =====  
===== First - using method calls =====  
Chapter: The command pattern.  
Section 1: Problem  
Command is a pattern that makes behavior an object.
```

```
---> Erasing last entered line  
Chapter: The command pattern.  
Section 1: Problem
```

```
===== Next - command objects assigned to F1..F3 =====  
Chapter: The command pattern.  
Section 1: Problem  
Command is a pattern that makes behavior an object.
```

```
===== F2 reassigned and pressed =====  
Chapter: The command pattern.  
Section 1: Problem  
Command is a pattern that makes behavior an object.  
A wrong line
```

```
===== Undo of last insert =====  
Chapter: The command pattern.  
Section 1: Problem  
Command is a pattern that makes behavior an object.
```

```
public static void main(String[] args) {  
    System.out.println( "===== Demonstration of Command =====");  
    Document doc = new Document();  
  
    String line1 = "Chapter: The command pattern.";  
    String line2 = "Section 1: Problem";  
    String line3 = "Command is a pattern that makes behavior an object.";  
  
    System.out.println( "===== First - using method calls =====" );  
    doc.write(line1);  
    doc.write(line2);  
    doc.write(line3);  
    System.out.println( doc );  
  
    System.out.println( "---> Erasing last entered line" );  
    doc.erase(line3);  
    System.out.println( doc );  
  
    System.out.println( "===== Next - command objects assigned to F1..F3 =====" );  
    doc = new Document();  
    // Create the commands  
    Command write1, write2, write3;  
    write1 = new WriteCommand(doc, line1);  
    write2 = new WriteCommand(doc, line2);  
    write3 = new WriteCommand(doc, line3);  
    // Note - nothing has happened to the document yet!  
  
    // assign bindings to the F1 keys  
    FKey F1 = new FKey(), F2 = new FKey(), F3 = new FKey();  
    // assign write commands to the keys  
    F1.assign(write1);  
    F2.assign(write2);  
    F3.assign(write3);  
  
    // next press F1 to F3 and see the result in the document  
    F1.press(); F2.press(); F3.press();  
    System.out.println( doc );  
  
    // reassigning F2  
    System.out.println( "===== F2 reassigned and pressed =====" );  
    Command write4 = new WriteCommand(doc, "A wrong line");  
    F2.assign(write4);  
    F2.press();  
    System.out.println( doc );  
  
    // undoing the last operation  
    System.out.println( "===== Undo of last insert =====" );  
    write4.undo();  
    System.out.println( doc );  
}
```



```
csdev@m1:~/proj/frsproject/command-demo$ java CommandDemo
===== Demonstration of Command =====
===== First - using method calls =====
Chapter: The command pattern.
Section 1: Problem
Command is a pattern that makes behavior an object.

---> Erasing last entered line
Chapter: The command pattern.
Section 1: Problem

===== Next - command objects assigned to F1..F3 =====
Chapter: The command pattern.
Section 1: Problem
Command is a pattern that makes behavior an object.

===== F2 reassigned and pressed =====
Chapter: The command pattern.
Section 1: Problem
Command is a pattern that makes behavior an object.
A wrong line

===== Undo of last insert =====
Chapter: The command pattern.
Section 1: Problem
Command is a pattern that makes behavior an object.
```

```
public static void main(String[] args) {
    System.out.println( "===== Demonstration of Command =====");
    Document doc = new Document();

    String line1 = "Chapter: The command pattern.";
    String line2 = "Section 1: Problem";
    String line3 = "Command is a pattern that makes behavior an object.";

    System.out.println( "===== First - using method calls =====" );
    doc.write(line1);
    doc.write(line2);
    doc.write(line3);
    System.out.println( doc );

    System.out.println( "---> Erasing last entered line" );
    doc.erase(line3);
    System.out.println( doc );

    System.out.println( "===== Next - command objects assigned to F1..F3 =====" );
    doc = new Document();
    // Create the commands
    Command write1, write2, write3;
    write1 = new WriteCommand(doc, line1);
    write2 = new WriteCommand(doc, line2);
    write3 = new WriteCommand(doc, line3);
    // Note - nothing has happened to the document yet!

    // assign bindings to the F1 keys
    FKey F1 = new FKey(), F2 = new FKey(), F3 = new FKey();
    // assign write commands to the keys
    F1.assign(write1);
    F2.assign(write2);
    F3.assign(write3);

    // next press F1 to F3 and see the result in the document
    F1.press(); F2.press(); F3.press();
    System.out.println( doc );

    // reassigning F2
    System.out.println( "===== F2 reassigned and pressed =====" );
    Command write4 = new WriteCommand(doc, "A wrong line");
    F2.assign(write4);
    F2.press();
    System.out.println( doc );

    // undoing the last operation
    System.out.println( "===== Undo of last insert =====" );
    write4.undo();
    System.out.println( doc );
}
```



```
csdev@m1:~/proj/frsproject/command-demo$ java CommandDemo
===== Demonstration of Command =====
===== First - using method calls =====
Chapter: The command pattern.
Section 1: Problem
Command is a pattern that makes behavior an object.

---> Erasing last entered line
Chapter: The command pattern.
Section 1: Problem

===== Next - command objects assigned to F1..F3 =====
Chapter: The command pattern.
Section 1: Problem
Command is a pattern that makes behavior an object.

===== F2 reassigned and pressed =====
Chapter: The command pattern.
Section 1: Problem
Command is a pattern that makes behavior an object.
A wrong line

===== Undo of last insert =====
Chapter: The command pattern.
Section 1: Problem
Command is a pattern that makes behavior an object.
```

```
public static void main(String[] args) {
    System.out.println( "===== Demonstration of Command =====");
    Document doc = new Document();

    String line1 = "Chapter: The command pattern.";
    String line2 = "Section 1: Problem";
    String line3 = "Command is a pattern that makes behavior an object.";

    System.out.println( "===== First - using method calls =====" );
    doc.write(line1);
    doc.write(line2);
    doc.write(line3);
    System.out.println( doc );

    System.out.println( "---> Erasing last entered line" );
    doc.erase(line3);
    System.out.println( doc );

    System.out.println( "===== Next - command objects assigned to F1..F3 =====" );
    doc = new Document();
    // Create the commands
    Command writel, write2, write3;
    writel = new WriteCommand(doc, line1);
    write2 = new WriteCommand(doc, line2);
    write3 = new WriteCommand(doc, line3);
    // Note - nothing has happened to the document yet!

    // assign bindings to the F1 keys
    FKey F1 = new FKey(), F2 = new FKey(), F3 = new FKey();
    // assign write commands to the keys
    F1.assign(writel);
    F2.assign(write2);
    F3.assign(write3);

    // next press F1 to F3 and see the result in the document
    F1.press(); F2.press(); F3.press();
    System.out.println( doc );

    // reassigning F2
    System.out.println( "===== F2 reassigned and pressed =====" );
    Command write4 = new WriteCommand(doc, "A wrong line");
    F2.assign(write4);
    F2.press();
    System.out.println( doc );

    // undoing the last operation
    System.out.println( "===== Undo of last insert =====" );
    write4.undo();
    System.out.println( doc );
}
```




```
csdev@m1:~/proj/frsproject/command-demo$ java CommandDemo
===== Demonstration of Command =====
===== First - using method calls =====
Chapter: The command pattern.
Section 1: Problem
Command is a pattern that makes behavior an object.

---> Erasing last entered line
Chapter: The command pattern.
Section 1: Problem

===== Next - command objects assigned to F1..F3 =====
Chapter: The command pattern.
Section 1: Problem
Command is a pattern that makes behavior an object.

===== F2 reassigned and pressed =====
Chapter: The command pattern.
Section 1: Problem
Command is a pattern that makes behavior an object.
A wrong line

===== Undo of last insert =====
Chapter: The command pattern.
Section 1: Problem
Command is a pattern that makes behavior an object.
```

```
public static void main(String[] args) {
    System.out.println( "===== Demonstration of Command =====");
    Document doc = new Document();

    String line1 = "Chapter: The command pattern.";
    String line2 = "Section 1: Problem";
    String line3 = "Command is a pattern that makes behavior an object.";

    System.out.println( "===== First - using method calls =====" );
    doc.write(line1);
    doc.write(line2);
    doc.write(line3);
    System.out.println( doc );

    System.out.println( "---> Erasing last entered line" );
    doc.erase(line3);
    System.out.println( doc );

    System.out.println( "===== Next - command objects assigned to F1..F3 =====" );
    doc = new Document();
    // Create the commands
    Command writel, write2, write3;
    writel = new WriteCommand(doc, line1);
    write2 = new WriteCommand(doc, line2);
    write3 = new WriteCommand(doc, line3);
    // Note - nothing has happened to the document yet!

    // assign bindings to the F1 keys
    FKey F1 = new FKey(), F2 = new FKey(), F3 = new FKey();
    // assign write commands to the keys
    F1.assign(writel);
    F2.assign(write2);
    F3.assign(write3);

    // next press F1 to F3 and see the result in the document
    F1.press(); F2.press(); F3.press();
    System.out.println( doc );

    // reassigning F2
    System.out.println( "===== F2 reassigned and pressed =====" );
    Command write4 = new WriteCommand(doc, "A wrong line");
    F2.assign(write4);
    F2.press();
    System.out.println( doc );

    // undoing the last operation
    System.out.println( "===== Undo of last insert =====" );
    write4.undo();
    System.out.println( doc );
}
```

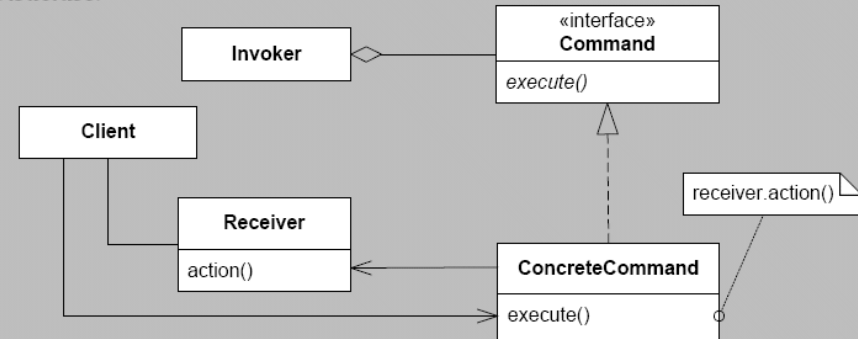
[23.1] Design Pattern: Command

Intent Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Problem You want to configure objects with behavior/actions at run-time and/or support undo.

Solution Instead of defining operations in terms of methods, define them in terms of objects implementing an interface with an `execute` method. This way requests can be associated to objects dynamically, stored and replayed, etc.

Structure:



Roles **Invoker** is an object, typically user interface related, that may execute a **Command**, that defines the responsibility of being an executable operation. **ConcreteCommand** defines the concrete operations that involves the object, **Receiver**, that the operation is intended to manipulate. The **Client** creates concrete commands and sets their receivers.

Cost - Benefit Objects that invoke operations are decoupled from those that know how to perform it. *Commands are first-class objects*, and can be manipulated like all other objects. You can assemble commands into *composite commands* (macros). It is *easy to add new commands*.

Consequences

- Benefits
 - Decouples clients from set of commands
 - Command set can be extended at run-time
 - Easy to support multiple ways to execute command (menu, pop up, shortcut key, tool bar, ...)
 - Commands are first-class objects
 - Log them, store them
 - Assembling macros is easy (composite of commands)
 - Undo can be supported
 - Add an 'unexecute()' method, and stack the set of executed commands.
- Liability: Cumbersome code for calling a method