

Chapter

23

Command

23.1 The Problem

Consider a graphical user interface that lets the user parameterize the meaning of short-cut keys and action buttons in the interface. For instance one user assigns the “open document” functionality to the “F1” button while another user assigns “save” to this button. That is, the same event, pressing the F1 key, should invoke different methods on potentially different objects. Consider another situation where a user finds himself repeating the same set of operations over and over again. He would like to be able to record the operations in a “macro” and then assign this macro to a short-cut key. A third scenario: a user has regretted the last three operations on the document, and wants to undo them.

All these requests are actually pretty difficult to handle because behavior is defined by object methods, and methods are not objects that can be stored or passed as parameters. Consider a method that is called whenever the user presses F1. How can I code this method in such a way that it is the user who decides if its contents is

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

I may come up with a parametric solution with a (very) long list of potential assignments of the F1 button. Still the list is hardcoded into the code and the user cannot assign any operation outside the list defined by the developer.

☞ Consider how to handle the macro and undo scenarios.

23.2 A Solution

The basic solution to the problem of methods not being objects—is to make objects that encapsulate the methods. The ③-①-② process applies:

- 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.


Thus the method call to save a document

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

becomes something like

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

Note that creating the “method” and execution of it is now two distinct steps, thus you can create the object in one place, and defer execution to another part of the code, say when a short-cut key is pressed.

 Study the source code provided in folder *chapter/command* on the web site.

Exercise 23.1: In the associated source code, the user assigns a new command to the F2 key:

Fragment: chapter/command/CommandDemo.java

```
Command write4 = new WriteCommand(doc, "A wrong line");  
F2.assign(write4);  
F2.press();
```

Draw a UML sequence diagram showing the interaction between the roles in the COMMAND pattern when the last statement executes.

23.3 The Command Pattern

This is the **COMMAND** pattern (design pattern box 23.1, page 36): objects that take on the single responsibility to represent a method. Its intent is

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

The central roles are the **Command** that defines the interface for execution which is then called by the **Invoker**. **ConcreteCommand** implements the **Command** interface and defines the concrete behavior. The concrete command objects must know the **Receiver**, that is the object to invoke the method on. A standard object-oriented method invocation has a receiver and a set of parameters

```
object.method(a,b,c);
```

Here `object` is the receiver, and therefore a command object must have the exact same set of parameters in order to be executable. Typically these parameters are set in the constructor:

```
Command method = new MethodCommand(object,a,b,c);
```

This creation of the command object and binding it to the **receiver** is the responsibility of the **Client** role. The client can then configure the invoker with the command object, and the invoker can later perform the method call:

```
method.execute();
```

As a developer can always create new classes implementing the **Command** interface the set of commands is not limited to the imagination of the original developers: it is *change by addition, not by modification*. And as the invoker only expects the **Command** role, all new commands can be handled exactly as the old ones.

While it is not mandatory, it is possible to add undo when using command objects. It of course requires that the command *can* be undone and that the receiver supports reversing the state changes from a given method call:

```
public class MethodCommand {  
    ...  
    public void execute() {  
        object.method(a,b,c);  
    }  
    public void undo() {  
        object.undoTheMethod(a,b,c);  
    }  
}
```

As requests are now objects, they can be manipulated in all the usual ways. For instance if all commands support undo, then all the executed commands in a session can be stored on a stack to support unlimited undo: just pop the next command object from the stack and invoke its `undo` method. You can store a list of commands on disk and recover from a crash by executing them in sequence. You can define a macro command as a **COMPOSITE** of **COMMAND** objects.

The major liability of COMMAND is the large overhead in writing and executing commands: compare the simple `object.method(a, b, c)`; call with all the extra typing to make command interfaces, concrete classes, and setting up the command object. Thus, COMMAND is a heavy weight approach that only should be used when the required flexibility makes the effort worthwhile.

23.4 Selected Solutions

Discussion of Exercise 23.1:

As shown in Figure 23.1 the sequence is rather simple, however the construction of the write command object is not drawn.

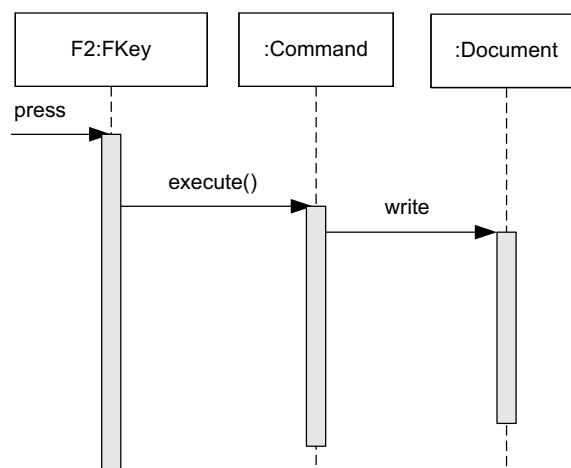


Figure 23.1: Executing a write command assigned to F2.

23.5 Review Questions

Describe the COMMAND pattern. What problem does it solve? What is its structure and what is the protocol? What roles and responsibilities are defined? What are the benefits and liabilities?

23.6 Further Exercises

Exercise 23.2. Source code directory:

`chapter/command`

The example code used in this chapter should be extended with a `MacroCommand` class whose responsibilities are

MacroCommand

- To record a set of commands (add commands to a list)
- To execute all commands in the list as a unit

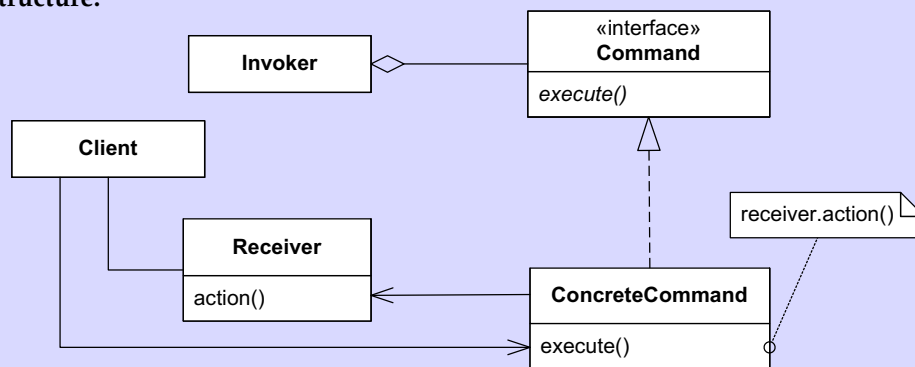
[23.1] Design Pattern: Command

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*.