InterBit
Training & Consulting Ltd.

Command Pattern

Chapter Content

InterBit
Training & Consulting Ltd.

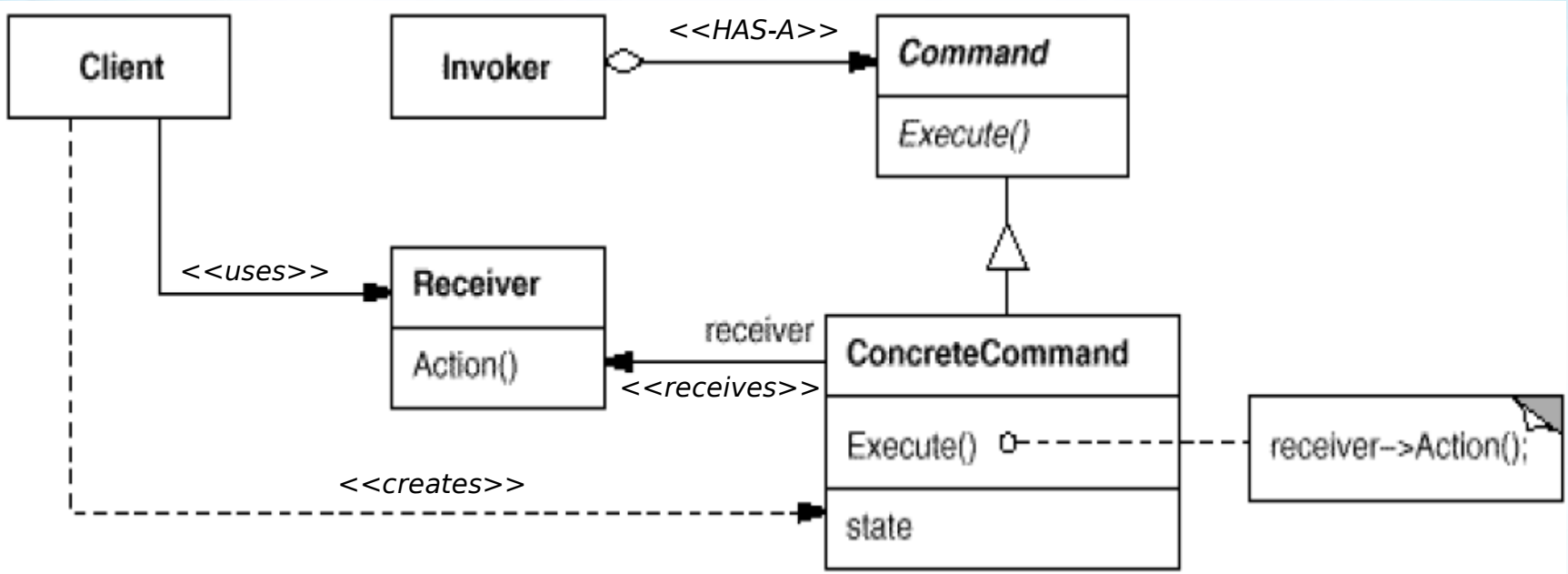
- The Command Pattern
- Command – UML Diagram
- Command Example
- Using our Commands
- SwingUtilities.invokeLater:

The Command Pattern

InterBit Training & Consulting Ltd.

- **A Command object represents some code to be executed.**
  - Other classes will be required to execute this command without caring what's inside:
    - Hence, flexibility & modularity.
- **C++** *Functors* fit this pattern.
  - But they also fit other patterns (shown later).
- OOP replacement for pointers to functions.

Command – UML Diagram



The classes participating in the pattern are:

- Command - declares an interface for executing an operation;

- ConcreteCommand - extends the Command interface, implementing the Execute method by invoking the corresponding operations on Receiver. It defines a link between the Receiver and the action.

- Client - creates a ConcreteCommand object and sets its receiver;

- Invoker - asks the command to carry out the request;

- Receiver - knows how to perform the operations;

Command Example

InterBit Training & Consulting Ltd.

```
interface Command {
    void execute();
}
class PrintCommand implements Command {

    private String str;

    public PrintCommand(String str) { this.str=str; }
    public void execute() {
       System.out.println(str);
    }
}
class DelCommand implements Command {

    private File file;

    public DelCommand(String fn) { file=new File(fn); }
    public void execute() {
        try {

           file.delete();

        }catch(Exception e) { e.printStackTrace();}

   }
}
```

Using our commands

InterBit Training & Consulting Ltd.

```
// Queue of commands to be executed:
class QueueManager {
    private ArrayList queue=new ArrayList();
    public synchronized void add(Command com){
     queue.add(com);
     this.notifyAll();
    }
    public synchronized void pop(){
     while(queue.size()==0) {
         try {
          this.wait();
         }catch(InterruptedException e){}
         Command com=(Command)queue.get(0);
         com.execute();
     }
    }
}
```

Command – cont.

InterBit Training & Consulting Ltd.

- This queue may be on a (very non-secure) server. Clients will use object streams to send in Commands for server to execute.
- **Technical note:**
  - Since command sub-classes can't change the *execute* signature, we may use class variables instead of parameters.
- **Support for Un-do:**
  - Consider adding an unexecute() method to your command , if possible.

SwingUtilities.invokeLater:

InterBit Training & Consulting Ltd.

```
public class InvokeTestFrame extends JFrame {
     private JTextArea timeFld = new JTextField(5);
     private JButton   goBt    = new JButton("Go ");
     public InvokeTestFrame(){
          ...
          goBt.addActionListener(new ActionListener(){
               public void actionPerformed(ActionEvent ev){
                    new CounterThread().start();
               }
          });
     }
     class CounterThread extends Thread{
         public void run(){
            for(int i=0; i<20; i++) {
                final int num = i;
                SwingUtilities.invokeLater( new Runnable(){public void run(){
                   timeFld.setText(num + "");
               }
                });
                try { Thread.sleep(1000); } catch(InterruptedException e){}
```

The intelligence of a command

- There are two extremes that a programmer must avoid when using this pattern:

1. The command is just a link between the receiver and the actions that carry out the request
2. The command implements everything itself, without sending anything to the receiver.

- We must always keep in mind the fact that the receiver is the one who knows how to perform the operations needed

- The purpose of the command being to help the client to delegate its request quickly and to make sure the command ends up where it should.

Undo and redo actions

InterBit Training & Consulting Ltd.

- Some implementations of the *Command* design pattern include parts for supporting undo and redo of actions
- In order to do that a mechanism to obtain past states of the Receiver object is needed; in order to achieve this there are two options:
  - Before running each command a snapshot of the receiver state is stored in memory (expensive)
  - Only a set of performed operations are stored in memory. In this case the command and receiver classes should implement the inverse algorithms to undo each action.

Asynchronous Method Invocation

InterBit Training & Consulting Ltd.

- Another usage for the command design pattern is to run commands asynchronous in background of an application:
- The invoker is running in the main thread and sends the requests to the receiver which is running in a separate thread
- The invoker will keep a queue of commands to be run and will send them to the receiver while it finishes running them
- Instead of using one thread in which the receiver is running more threads can be created for this.

Adding new commands

InterBit Training & Consulting Ltd.

- The command object decouples the object that invokes the action from the object that performs the action
- There are implementations of the pattern in which the invoker instantiates the concrete command objects.
  - In this case if we need to add a new command type we need to change the invoker as well.
  - This would violate the Open Close Principle (OCP)
- In order to have the ability to add new commands with minimum of effort we have to make sure that the invoker is aware only about the abstract command class or interface.

Using composite commands

InterBit Training & Consulting Ltd.

- When adding new commands to the application we can use the composite pattern to group existing commands in another new command.
- This way, macros can be created from existing commands.

Command Summary

InterBit
Training & Consulting Ltd.

- The main advantage of the command design pattern is that it decouples the object that invokes the operation from the one that know how to perform it.
- This advantage must be kept.
- There are implementations of this design pattern in which the invoker is aware of the concrete commands classes.
- This is wrong making the implementation more tightly coupled. The invoker should be aware only about the abstract command class.