References

Design Patterns: Elements of Resuable Object-Oriented Software,
Gamma, Helm, Johnson, Vlissides, Addison-Wesley, 1995, pp. 175-184, 233-242

Pattern-Oriented Software Architecture: A System of Patterns, Buschman, Meunier, Rohnert,
Sommerlad, Stal, 1996, pp. 277-290, Command Processor

Command Processor, Sommerlad in Pattern Languages of Program Design 2, Eds. Vlissides,
Coplien, Kerth, Addison-Wesley, 1996, pp. 63-74

Photographs used with permission from www.istockphoto.com
Decorator
Adds responsibilities to individual objects

Dynamically

Transparenly
import java.io.*;
import sdsu.io.*;
class ReadingFileExample
{
    public static void main( String args[] ) throws Exception
    {
        FileInputStream inputFile;
        BufferedInputStream bufferedFile;
        ASCIIInputStream cin;
        inputFile = new FileInputStream( "ReadingFileExample.java" );
        bufferedFile = new BufferedInputStream( inputFile );
        cin = new ASCIIInputStream( bufferedFile );
        inputFile = new FileInputStream( "ReadingFileExample.java" );
        bufferedFile = new BufferedInputStream( inputFile );
        cin = new ASCIIInputStream( bufferedFile );
    }
}
Decorator forwards all component operations
Favor Composition over Inheritance
Refactoring: Move Embellishment to Decorator

Client → aBinaryTree.toArray

Client → anOddValueDecorator.toArray → aBinaryTree toArray
Benefits & Liabilities

Benefits

Simplifies a class
Distinguishes a classes core responsibilities from embellishments

Liabilities

Changes the object identity of a decorated object
Code harder to understand and debug
Combinations of decorators may not work correctly together
Command
Command

Encapsulates a request as an object

Example

Invoker be a menu
Client be a word processing program
Receiver a document
Action be save
When to Use the Command Pattern

Need action as a parameter (replaces callback functions)

Specify, queue, and execute requests at different times

Undo

Logging changes

High-level operations built on primitive operations

A transaction encapsulates a set of changes to data

Systems that use transaction often can use the command pattern

Macro language
Callback Function vs Command

Command contains reference to object that it acts on
Consequences

Command decouples the object that invokes the operation from the one that knows how to perform it.

It is easy to add new commands, because you do not have to change existing classes.

You can assemble commands into a composite object.
Refactoring: Replace Conditional Dispatcher with Command

public class SDSUChatServer {
    public void processClientRequest(String request) {
        blah
        if (command.equals("quit"))
            quit();
        else if (command.equals("register"))
            registerNewUser(commandData);
        else if (command.equals("login"))
            login(commandData);
        else if (command.equals("nickname"))
            checkNickname(commandData);
        blah
    }
    // replaced with action = actions.get(command);
    // action.execute(commandData);
}
public class RegisterCommand extends Command {
    private SDSUChatServer target;

    public RegisterCommand(SDSUChatServer aServer) {
        target = aServer;
    }

    public void execute(String commandData) {
        target.registerNewUser(commandData);
    }
}
public class SDSUChatServer {
    private HashMap<String, Command> actions;

    private populateActions() {
        actions = new HashMap<String, Command>();
        actions.put("quit", new QuitCommand(this));
        actions.put("register", new RegisterCommand(this));
        actions.put("login", new LoginCommand(this));
        actions.put("nickname", new NicknameCommand(this));
    }
}
When to do this?

Need runtime flexibility

Conditional Dispatcher is bloated
Pluggable Commands

- Can create one general Command using reflection
- Don’t hard code the method called in the command
- Pass the method to call an argument
import java.util.*;
import java.lang.reflect.*;

public class Command
{
    private Object receiver;
    private Method command;
    private Object[] arguments;

    public Command(Object receiver, Method command,
                    Object[] arguments )
    {
        this.receiver = receiver;
        this.command = command;
        this.arguments = arguments;
    }

    public void execute() throws InvocationTargetException,
                                IllegalAccessException
    {
        command.invoke( receiver, arguments );
    }
}
public class Test {
    public static void main(String[] args) throws Exception {
        Vector sample = new Vector();
        Class[] argumentTypes = { Object.class };
        Method add =
            Vector.class.getMethod( "addElement", argumentTypes);
        Object[] arguments = { "cat" };

        Command test = new Command(sample, add, arguments );
        test.execute();
        System.out.println( sample.elementAt( 0));
    }
}
Command Processor Pattern
Command Processor Pattern

Command Processor manages the command objects

The command processor:

Contains all command objects

Schedules the execution of commands

May store the commands for later unto

May log the sequence of commands for testing purposes

Uses singleton to insure only one instance
Structure

Command Processor
- `dolt(command)`
- `undolt()`
- `commandStack`

Command
- `execute()`

Receiver
- `action()`

ConcreteCommand
- `execute()`
- `receiver`

Client
- `transfer command`
- `creates`

Command Processor performs stores

Command performs

ConcreteCommand creates

receiver->action()
Dynamics

- Client
  - request
    - create()
    - undo()
      - undoIt()
      - undo()
    - delete()
    - restoreText()
    - getSelection()
    - makeBold()
    - do()
      - doIt()
    - create()
      - makeBold
      - command

- Command Processor
  - makeBold
  - command

- Document
  - getSelection()
  - makeBold()
  - restoreText()
  - delete()
**Benefits**

Flexibility in the way requests are activated

Different user interface elements can generate the same kind of command object

Allows the user to configure commands performed by a user interface element

Flexibility in the number and functionality of requests

Adding new commands and providing for a macro language comes easy

Programming execution-related services

Commands can be stored for later replay

Commands can be logged

Commands can be rolled back

Testability at application level

Concurrency

Allows for the execution of commands in separate threads
Liabilities

Efficiency loss

Potential for an excessive number of command classes

Try reducing the number of command classes by:

Grouping commands around abstractions
Unifying simple commands classes by passing the receiver object as a parameter

Complexity

How do commands get additional parameters they need?