Pearson New International Edition

Agile Software Development, Principles, Patterns, and Practices

Robert C. Martin

First Edition

ALWAYS LEARNING

PEARSON

Pearson New International Edition

Agile Software Development,
Principles, Patterns, and Practices
Robert C. Martin
First Edition

PEARSON

ate Command. This puts all the *wiring*[1] in one place and gets it out of the main body of the system. Indeed, it would be possible to create a simple text file that described which Sensors were bound to which Commands. The initialization program could read this file and build the system appropriately. Thus, the *wiring* of the system could be determined completely outside the program, and it could be adjusted without recompilation.

By encapsulating the *notion* of a command, this pattern allowed us to decouple the logical interconnections of the system from the devices that were being connected. This was a huge benefit.

Transactions

Another common use of the COMMAND pattern, and one that we will find useful in the Payroll problem, is in the creation and execution of transactions. Imagine, for example, that we are writing the software that maintains a database of employees. (See Figure 13-4.) There are a number of operations that users can apply to that database. They can add new employees, delete old employees, or change the attributes of existing employees.

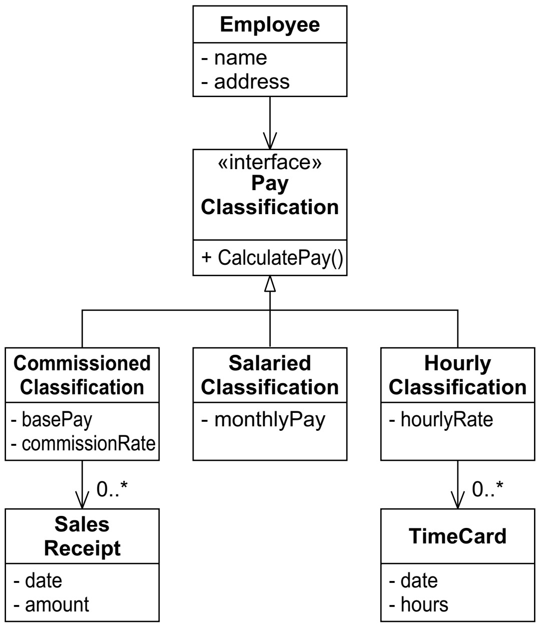

Figure 13-4 Employee Database

When a user decides to add a new employee, that user must specify all the information needed to successfully create the employee record. Before acting on that information, the system needs to verify that the information is syntactically and semantically correct. The COMMAND pattern can help with this job. The command object acts as a repository for the unvalidated data, implements the validation methods, and implements the methods that finally execute the transaction.

For example, consider Figure 13-5. The AddEmployeeTransaction contains the same data fields that Employee contains. It also holds a pointer to a PayClassification object. These fields and object are created from the data that the user specifies when directing the system to add a new employee.

The validate method looks over all the data and makes sure it makes sense. It checks it for syntactic and semantic correctness. It may even check to ensure that the data in the transaction are consistent with the existing state of the database. For example, it might ensure that no such employee already exists.

1. The logical interconnections between the Sensors and Commands.

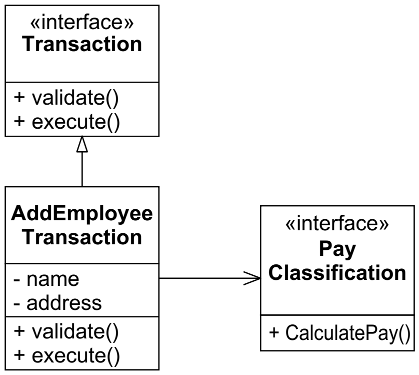

Figure 13-5 `AddEmployee` Transaction

The `execute` method uses the validated data to update the database. In our simple example, a new `Employee` object would be created and loaded with the fields from the `AddEmployeeTransaction` object. The `PayClassification` object would be moved or copied into the `Employee`.

Physical and Temporal Decoupling

The benefit this gives us is in the dramatic decoupling between the code that procures the data from the user, the code that validates and operates on that data, and the business objects themselves. For example, one might expect the data for adding a new employee to be procured from a dialog box in some GUI. It would be a shame if the GUI code contained the validation and execution algorithms for the transaction. Such a coupling would prevent that validation and execution code from being used with other interfaces. By separating the validation and execution code into the `AddEmployeeTransaction` class, we have physically decoupled that code from the procurement interface. What's more, we've separated the code that knows how to manipulate the logistics of the database from the business entities themselves.

Temporal Decoupling

We have also decoupled the validation and execution code in a different way. Once the data have been procured, there is no reason why the validation and execution methods must be called immediately. The transaction objects can be held in a list and validated and executed much later.

Suppose we have a database that must remain unchanged during the day. Changes may only be applied during the hours between midnight and 1 A.M. It would be a shame to have to wait until midnight and then have to rush to type in all the commands before 1 A.M. It would be much more convenient to type in all the commands, have them validated on the spot, and then executed later at midnight. The COMMAND pattern gives us this ability.

UNDO

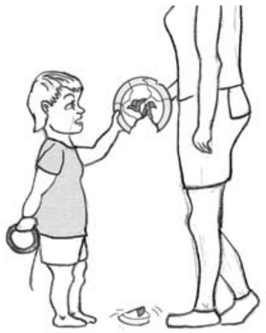

Figure 13-6 adds the `undo()` method to the COMMAND pattern. It stands to reason that if the `do()` method of a `Command` derivative can be implemented to remember the details of the operation it performs, then the `undo()` method can be implemented to undo that operation and return the system to its original state.

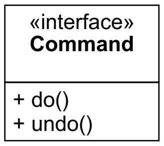

Figure 13-6 Undo variation of the COMMAND Pattern.

Imagine, for example, an application that allows the user to draw geometric shapes on the screen. A toolbar has buttons that allow the user to draw circles, squares, rectangles, etc. Let's say that the user clicks on the *draw circle* button. The system creates a `DrawCircleCommand` and then calls `do()` on that command. The `DrawCircleCommand` object tracks the user's mouse waiting for a click in the drawing window. On receiving that click, it sets the click point as the center of the circle and proceeds to draw an animated circle at that center with a radius that tracks the current mouse position. When the user clicks again, the `DrawCircleCommand` stops animating the circle and adds the appropriate circle object to the list of shapes currently displayed on the canvas. It also stores the ID of the new circle in some private variable of its own. Then it returns from the `do()` method. The system then pushes the expended `DrawCirlceCommand` on the stack of completed commands.

Later, the user clicks the undo button on the toolbar. The system pops the completed commands stack and calls `undo()` on the resulting `Command` object. on receiving the `undo()` message, the `DrawCircleCommand` object deletes the circle matching the saved ID from the list of objects currently displayed on the canvas.

With this technique, you can easily implement the `undo` command in nearly any application. The code that knows how to undo a command is always right next to the code that knows how to perform the command.

ACTIVE OBJECT

One of my favorite uses of the COMMAND pattern is the ACTIVE OBJECT pattern.[2] This is a very old technique for implementing multiple threads of control. It has been used, in one form or another, to provide a simple multitasking nucleus for thousands of industrial systems.

The idea is very simple. Consider Listings 13-2 and 13-3. An `ActiveObjectEngine` object maintains a linked list of `Command` objects. Users can add new commands to the engine, or they can call `run()`. The `run()` function simply walks through the linked list executing and removing each command.

Listing 13-2

`ActiveObjectEngine.java`

```
import java.util.LinkedList;
import java.util.Iterator;

public class ActiveObjectEngine
{
  LinkedList itsCommands = new LinkedList();

  public void addCommand(Command c)
  {
    itsCommands.add(c);
  }

  public void run()
  {
    while (!itsCommands.isEmpty())
    {
      Command c = (Command) itsCommands.getFirst();
      itsCommands.removeFirst();
      c.execute();
    }
  }
}
```

2. [Lavender96].

Listing 13-3

`Command.java`

```
public interface Command
{
  public void execute() throws Exception;
}
```

This may not seem very impressive. But imagine what would happen if one of the `Command` objects in the linked list cloned itself and then put the clone back on the list. The list would never go empty, and the `run()` function would never return.

Consider the test case in Listing 13-4. It creates something called a `SleepCommand`. Among other things, it passes a delay of 1000 ms to the constructor of the `SleepCommand`. It then puts the `SleepCommand` into the `ActiveObjectEngine`. After calling `run()`, it expects that a certain number of milliseconds has elapsed.

Listing 13-4

`TestSleepCommand.java`

```
import junit.framework.*;
import junit.swingui.TestRunner;

public class TestSleepCommand extends TestCase
{
  public static void main(String[] args)
  {
    TestRunner.main(new String[]{"TestSleepCommand"});
  }

  public TestSleepCommand(String name)
  {
    super(name);
  }

  private boolean commandExecuted = false;

  public void testSleep() throws Exception
  {
    Command wakeup = new Command()
    {
      public void execute() {commandExecuted = true;}
    };
    ActiveObjectEngine e = new ActiveObjectEngine();
    SleepCommand c = new SleepCommand(1000,e,wakeup);
    e.addCommand(c);
    long start = System.currentTimeMillis();
    e.run();
    long stop = System.currentTimeMillis();
    long sleepTime = (stop-start);
    assert("SleepTime " + sleepTime + " expected > 1000",
           sleepTime > 1000);
    assert("SleepTime " + sleepTime + " expected < 1100",
            sleepTime < 1100);
    assert("Command Executed", commandExecuted);
  }
}
```

Let's look at this test case more closely. The constructor of the `SleepCommand` contains three arguments. The first is the delay time in milliseconds. The second is the `ActiveObjectEngine` that the command will be running in. Finally, there is another command object called `wakeup`. The intent is that the `SleepCommand` will wait for the specified number of milliseconds and will then execute the `wakeup` command.

Listing 13-5 shows the implementation of `SleepCommand`. On execution, `SleepCommand` checks to see if it has been executed previously. If not, then it records the start time. If the delay time has not passed, it puts itself back in the `ActiveObjectEngine`. If the delay time has passed, it puts the `wakeup` command into the `ActiveObjectEngine`.

Listing 13-5

`SleepCommand.java`

```
public class SleepCommand implements Command
{
  private Command wakeupCommand = null;
  private ActiveObjectEngine engine = null;
  private long sleepTime = 0;
  private long startTime = 0;
  private boolean started = false;

  public SleepCommand(long milliseconds, ActiveObjectEngine e,
                      Command wakeupCommand)
  {
    sleepTime = milliseconds;
    engine = e;
    this.wakeupCommand = wakeupCommand;
  }

  public void execute() throws Exception
  {
    long currentTime = System.currentTimeMillis();
    if (!started)
    {
      started = true;
      startTime = currentTime;
      engine.addCommand(this);
    }
    else if ((currentTime - startTime) < sleepTime)
    {
      engine.addCommand(this);
    }
    else
    {
      engine.addCommand(wakeupCommand);
    }
  }
}
```

We can draw an analogy between this program and a multithreaded program that is waiting for an event. When a thread in a multithreaded program waits for an event, it usually invokes some operating system call that blocks the thread until the event has occurred. The program in Listing 13-5 does not block. Instead, if the event it is waiting for `((currentTime - startTime) < sleepTime)` has not occurred, it simply puts itself back into the `ActiveObjectEngine`.

Building multithreaded systems using variations of this technique has been, and will continue to be, a very common practice. Threads of this kind have been known as *run-to-completion* tasks (RTC), because each `Command` instance runs to completion before the next `Command` instance can run. The name RTC implies that the `Command` instances do not block.

The fact that the `Command` instances all run to completion gives RTC threads the interesting advantage that they all share the same run-time stack. Unlike the threads in a traditional multithreaded system, it is not necessary to define or allocate a separate run-time stack for each RTC thread. This can be a powerful advantage in memory-constrained systems with many threads.

Continuing our example, Listing 13-6 shows a simple program that makes use of `SleepCommand` and exhibits multithreaded behavior. This program is called `DelayedTyper`.

Listing 13-6

`DelayedTyper.java`

```
public class DelayedTyper implements Command
{
  private long itsDelay;
  private char itsChar;
  private static ActiveObjectEngine engine =
    new ActiveObjectEngine();
  private static boolean stop = false;

  public static void main(String args[])
  {
    engine.addCommand(new DelayedTyper(100,'1'));
    engine.addCommand(new DelayedTyper(300,'3'));
    engine.addCommand(new DelayedTyper(500,'5'));
    engine.addCommand(new DelayedTyper(700,'7'));

    Command stopCommand = new Command()
    {
      public void execute() {stop=true;}
    };

    engine.addCommand(
      new SleepCommand(20000,engine,stopCommand));
    engine.run();
  }

  public DelayedTyper(long delay, char c)
  {
    itsDelay = delay;
    itsChar = c;
  }

  public void execute() throws Exception
  {
    System.out.print(itsChar);
    if (!stop)
      delayAndRepeat();
  }
```