Software Development (CS2500)

Lecture 52: More Design Patterns

M.R.C. van Dongen

March 2, 2011

Contents

1 Outline

These lectures studies three *Design Patterns*.

Singleton pattern: Lets you define a class that's instantiated once.

Command pattern: Lets you encapsulate a request as an object.

Adapter pattern: Converts a incompatible interface to a compatible interface.

This lecture is based on [Gamma *et al.*, 2008; Bloch, 2008].

2 The Singleton Pattern

Some classes should be instantiated only once. For example, there should be only one file system, only one window manager, ... A class which is instantiated ony once is called a *singleton* classes.

Design Principle 1 (Singleton Pattern). *The* Singleton Pattern *lets you create classes that can be intantiated only once.*

The following is a typical implementation. This example is from [Bloch, 2008, Item 3].

```Java
public class Elvis {
    private static final Elvis INSTANCE = new Elvis( );

    private Elvis( ) { … }

    public static Elvis getInstance( ) { return INSTANCE; }
    public void leaveTheBuilding( ) { … }
}
```

The `final` guarantees that `INSTANCE` is instantiated only once. The `static` ensures that `INSTANCE` is a class attribute: there is only one of it. The `private` constructor guarantees that the constructor cannot be accessed from outside. In short this class seems to satisfy all requirements of a singleton class. (Still it is possible for an adverse client to invoke the constructor. This involves *reflection*, which we haven't covered yet.)

There turns out to be a better implementation of our `Elvis` class. This is the best way to implement a singleton class [Bloch, 2008, Item 3].

```Java
public enum Elvis {
    INSTANCE;

    public void leaveTheBuilding( ) { … }
}
```

This section studies the Command Pattern.

Design Principle 2 (Command Pattern). *The* Command Pattern *encapsulates a request as an object, thereby letting you parameterise clients with different requests, queue or log requests, and support undoable operations.*

To understand the pattern, let's carry out a case study. Many applications require an on-off facility.

On: Turn the light on, turn the television on, ...

Off: Turn the light off, turn the television off, ...

Undo: Optionally, undo the last operation.

Since on-off applications are so common, we want to handle them without knowing all their details. Let's implement a `OnOfSwitch` class for controlling on-off applications. The class has the following functionality:

`on( ):` Turn the application on: runs the on *command*.

`off( ):` Turn the application off: runs the off *command*.

`setOnCommand( ):` Set *the command* for `on( )`.

`setOffCommand( ):` Set *the command* for `off( )`.

To implement this we need a *Command interface*. We'll forget about the undo option.

In our `Command` interface we only define one method: `execute( )`. This method just carries out the required task. By creating concrete class instances we can carry out specific tasks.

```Java
public interface Command {
    public void execute( );
}
```

If on and off operations requires much coding effort the you probably want to implement a concrete class that executes the operation.

```Java
public class ConcreteTvOnCommand implements Command {
    public void execute( ) { System.out.println( "TV goes on." ); }
}
```

Otherwise, you probably want to implement a concrete `Command` instance with an inner class or an anonymous class.

```Java
Command tvOnCommand = new Command( ) {
    public void execute( ) { System.out.println( "TV goes on." ); }
};
```

The following is the `OnOffSwitch` class. The class issues the `execute( )` methods of the `Command` instances.

```Java
public class OnOffSwitch {
    private Command on;
    private Command off;

    public OnOffSwitch( Command onCommand, Command offCommand ) {
        this.on  = onCommand;
        this.off = offCommand;
    }

    public void on( ) {
        on.execute( );
    }

    public void off( ) {
        off.execute( );
    }

    public void setOnCommand( Command command ) {
        this.on = command;
    }

    public void setOffCommand( Command command ) {
        this.off = command;
    }
}
```

Let's use our `OnOfSwitch` class to control `Tv` and a `Light` objects. We'll use the following two classes to implement `Tv` and `Light` objects. Notice that the two classes are completely unrelated.

```Java
public class Tv {
    public void tvOn( )  { System.out.println( "Turning Tv on." ); }
    public void tvOff( ) { System.out.println( "Turning Tv off." ); }
}
```

```Java
public class Light {
    public void lightOn( )  { System.out.println( "Turning light on." ); }
    public void lightOff( ) { System.out.println( "Turning light off." ); }
}
```

The following shows how we may use the `OnOffSwitch` class. The implementation of the `private` methods is provided further on.

```Java
public class Client {
    public static void main( String[] args ) {
        OnOffSwitch tvSwitch = createInvoker( new Tv( ) );
        tvSwitch.on( );
        tvSwitch.off( );
        OnOffSwitch lightSwitch = createInvoker( new Light( ) );
        lightSwitch.on( );
        lightSwitch.off( );
    }

    private static OnOffSwitch createInvoker( final Tv tv ) {
        // Omitted
        return new OnOffSwitch( onCommand, offCommand );
    }

    private static OnOffSwitch createInvoker( final Light light ) {
        // Omitted
        return new OnOffSwitch( onCommand, offCommand );
    }
}
```

```Java
private static OnOffSwitch createInvoker( final Tv tv ) {
    Command onCommand = new Command( ) {
        private final Tv receiver = tv;
        public void execute( ) { receiver.tvOn( ); }
    };
    Command offCommand = new Command( ) {
        private final Tv receiver = tv;
        public void execute( ) { receiver.tvOff( ); }
    };
    return new OnOffSwitch( onCommand, offCommand );
}

private static OnOffSwitch createInvoker( final Light light ) {
    Command onCommand = new Command( ) {
        private final Light receiver = light;
        public void execute( ) { receiver.lightOn( ); }
    };
    Command offCommand = new Command( ) {
        private final Light receiver = light;
        public void execute( ) { receiver.lightOff( ); }
    };
    return new OnOffSwitch( onCommand, offCommand );
}
```

The general picture of the Command Pattern is as follows. The `Command` interface defines an interface for executing commands. Concrete `Command` classes define a *receiver* and an *action*. The action may not always be explicitly represented. Figure 1 depicts the Command Pattern graphically.

The following are some possible applications of the Command Pattern.

- Parameterise objects by an action to perform.
- Specify, queue, and execute command requests at different times. For example, you may create a

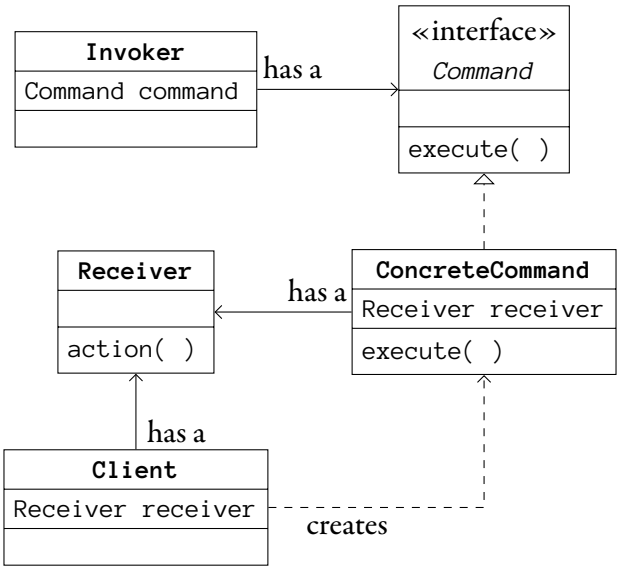

Figure 1: The `Command` Pattern.

`Command` object, send it to a receiver (processor), and request that the receiver execute the `Command`.

- Support `undo( )` operations. It is left as an exercise to implement this.
- Support logging changes. This allows you to replay a sequence of commands, recover from crashes, and so on.

3 The Adapter Pattern

This section studies the Adapter Pattern.

Design Principle 3 (Adapter Pattern). *The* Adapter Pattern *converts an incompatible interface into a compatible interface.*

Figure 2 depicts a client class called `DrawClass` and an interface called `Graphics`. The client class defines a method called `rect( )` which draws a rectangle by calling the instance method `drawRect( )` of a `Graphics` object instance variable which is called `g`. The `Graphics` interfaces defines this method but without a class implementing the interface, the `DrawClass` is useless.

The `Graphics` interface and the class `DrawClass` are *incompatible*, just like a European electricity socket and an Irish electricity plug are incompatible: your iPod charger can't be plugged into the European sockets. This is why most Irish people travelling to the continent take an adapter with them.[1] The adapter plugs in to the European electricity socket and their iPod charger plugs into the adapter.

The *Adapter Design Pattern* works just like the electricty adapter. It makes an interface compatible with a class by adapting it. Getting back to our example, we can implement an adapter for the `Graphics`

[1] That is to say, if they were thinking when they were packing their bags.

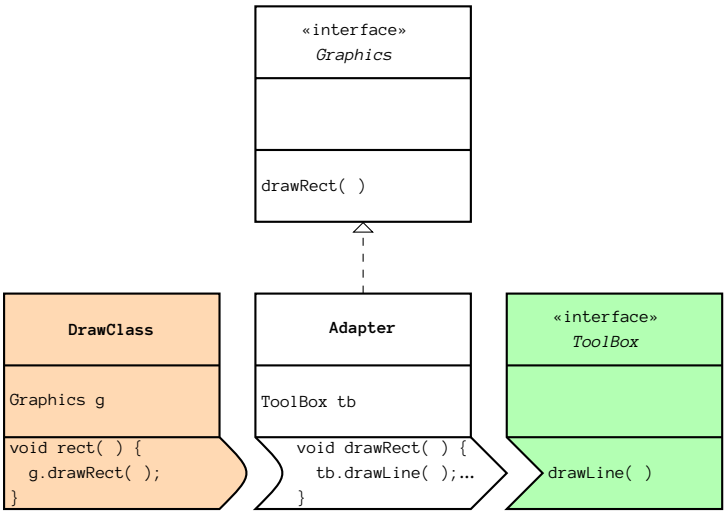

Figure 2: Incompatible interface.

interface by implementing a class called `Adapter` which implements `Graphics`. The `Adapter` class overrides `drawRect( )` and implements it using the method `drawLine( )`, which is defined in the interface `ToolBox`, which is known to have an implementation.

To complete the design, we initialise the instance variable `g` in the `DrawClass` as follows: `g = new Adapter( )`.

4 For Friday

Study the lecture notes.

5 Acknowledgements

This lecture is based on [Gamma *et al.*, 2008] and on [Bloch, 2008].

References

[Bloch, 2008] Joshua Bloch. *Effective Java*. Addison–Wesley, 2008.

[Gamma *et al.*, 2008] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns* Elements of Reusable Object-Oriented Software. Addison–Wesley, 2008. 36th Printing.

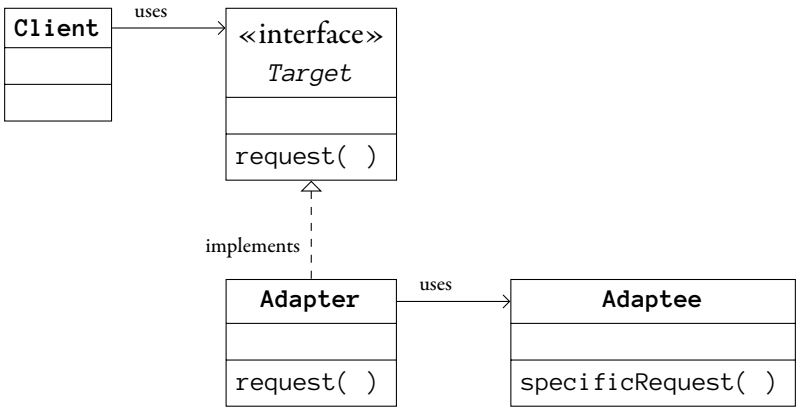

Figure 3: Delegation-based Adapter Pattern in UML.