Design Patterns: Part 2

ENGI 5895: Software Design

Andrew Vardy
with code samples from Dr. Rodrigue Byrne and [Martin(2003)]

Faculty of Engineering & Applied Science
Memorial University of Newfoundland

February 5, 2018

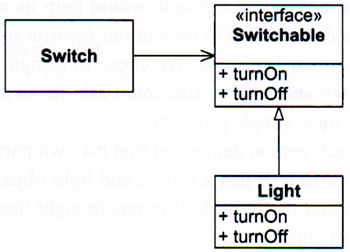
Outline

Adapter

Adapter converts the interface of a class into another form. For example, a Switch is used to control a Light. To adhere to the DIP and OCP we introduce a Switchable interface for various devices that can be switched on and off:

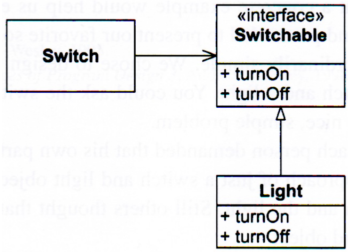
Adapter

Adapter converts the interface of a class into another form. For example, a Switch is used to control a Light. To adhere to the DIP and OCP we introduce a Switchable interface for various devices that can be switched on and off:



Adapter

Adapter converts the interface of a class into another form. For example, a Switch is used to control a Light. To adhere to the DIP and OCP we introduce a Switchable interface for various devices that can be switched on and off:



However, perhaps Light is provided by a third party and has only a toggle method. We need a class to translate between calls to turnOn and turnOff and calls to toggle.

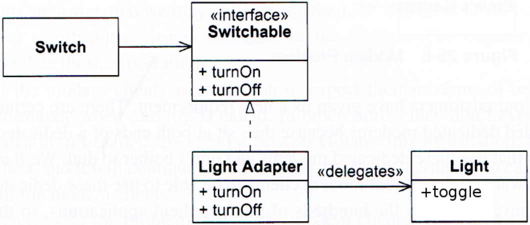


Figure 25-4 Solving the Table Lamp with ADAPTER

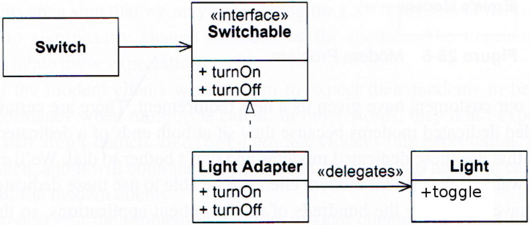


Figure 25-4 Solving the Table Lamp with ADAPTER

Notice the stereotype <<delegates>>. The responsibility of actually controlling the Light is **delegated** to Light. If you delegate a job, it means you are not doing it yourself.

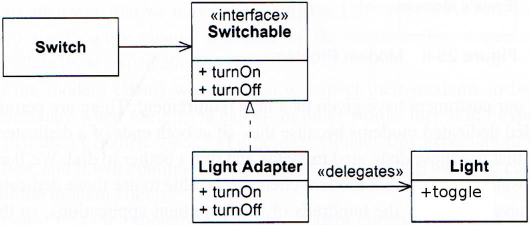


Figure 25-4 Solving the Table Lamp with ADAPTER

Notice the stereotype <<delegates>>. The responsibility of actually controlling the Light is **delegated** to Light. If you delegate a job, it means you are not doing it yourself.

In this example, LightAdapter has an associated Light. This is the **object form of adapter**. There is also a class form...

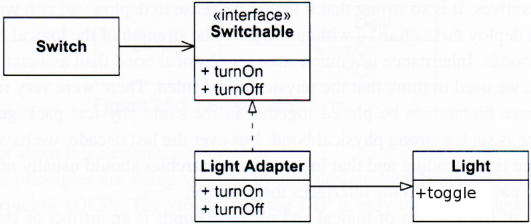


Figure 25-5 Solving the Table Lamp with ADAPTER

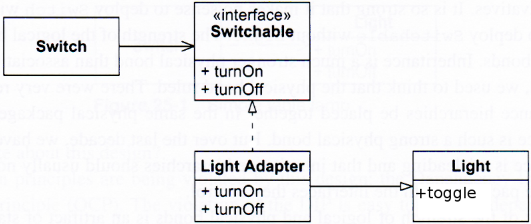


Figure 25-5 Solving the Table Lamp with ADAPTER

This is the class form of adapter. The only difference is whether the Adapter class inherits from Light or "has a" Light.

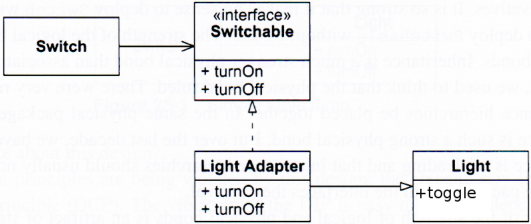


Figure 25-5 Solving the Table Lamp with ADAPTER

This is the class form of adapter. The only difference is whether the Adapter class inherits from Light or "has a" Light.

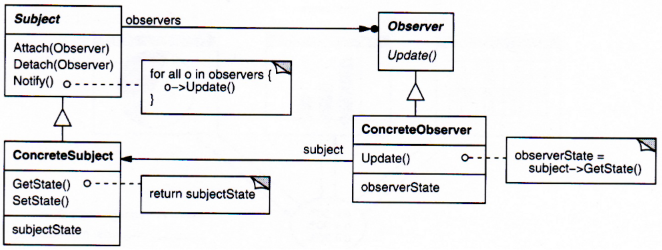
Inheritance is slightly easier since LightAdapter will not need a pointer to Light. However, inheritance forever binds LightAdapter to Light. It may be the case that we can re-use LightAdapter in another situation. In this case, we would prefer association over inheritance.

Observer

We may have one or more objects that need to update whenever some activity occurs in a subject object. The idea is for these observing or listening objects to subscribe to the subject. The subject will then automatically notify the observer objects to let them know that something has changed.

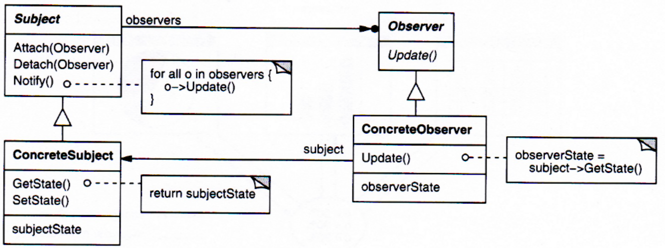
Observer

We may have one or more objects that need to update whenever some activity occurs in a subject object. The idea is for these observing or listening objects to subscribe to the subject. The subject will then automatically notify the observer objects to let them know that something has changed.



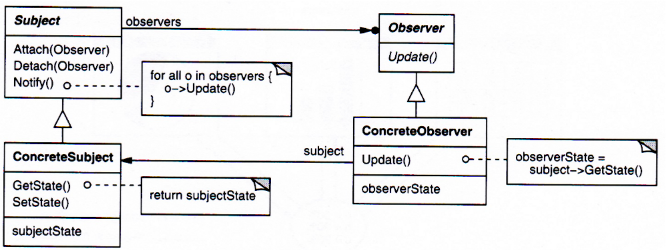
Observer

We may have one or more objects that need to update whenever some activity occurs in a subject object. The idea is for these observing or listening objects to subscribe to the subject. The subject will then automatically notify the observer objects to let them know that something has changed.

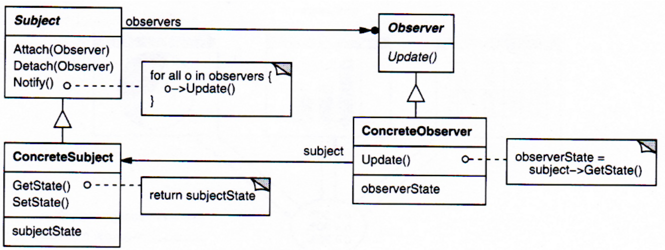


Classes which are the subject of observation should extend Subject. Observers should implement Observer!

[Figure repeated]

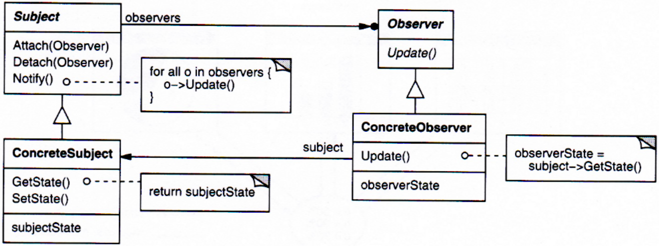


[Figure repeated]



A class such as Subject should either be abstract (some implementation allowed) or a pure interface (no implementation). It is often convenient for Subject to be abstract and provide implementations for methods such as Attach and Notify.

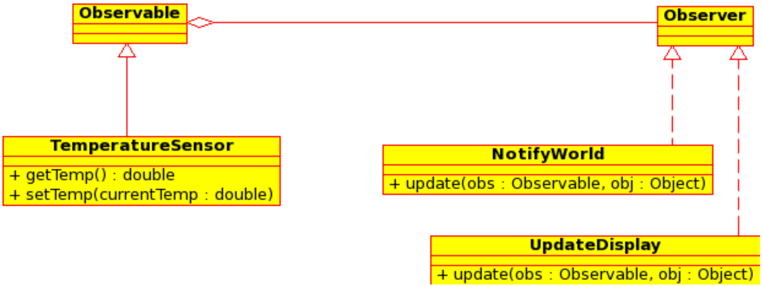
[Figure repeated]



A class such as Subject should either be abstract (some implementation allowed) or a pure interface (no implementation). It is often convenient for Subject to be abstract and provide implementations for methods such as Attach and Notify.

The designers of Java were so taken with Observer that they provided classes called java.util.Observable (i.e. Subject) and java.util.Observer. They also use Observer for Java's GUI event processing model (as we will see).

e.g. Temperature Observer



```java
import java.util.Observable;
class TemperatureSensor extends Observable {
    private double currentTemp;

    public double getTemp() {
        return currentTemp;
    }

    public void setTemp( double currentTemp ) {
        if ( this.currentTemp != currentTemp ) {
            this.currentTemp = currentTemp;
            setChanged(); // setChanged is protected
            notifyObservers();
        }
    }
}
```

```
import java.util.Observer;
import java.util.Observable;
class NotifyWorld implements Observer {
    public void update(Observable obs, Object obj) {
        TemperatureSensor sens = (TemperatureSensor)obs;
        System.out.printf("World new temp is %g%n",
            sens.getTemp() );
    }
}
```

```
import java.util.Observer;
import java.util.Observable;
class NotifyWorld implements Observer  {
    public void update(Observable obs, Object obj) {
        TemperatureSensor sens = (TemperatureSensor)obs;
        System.out.printf("World new temp is %g%n",
            sens.getTemp() );
    }
}

import java.util.Observable;
import java.util.Observer;

class UpdateDisplay implements Observer  {
    public void update(Observable obs, Object obj) {
        TemperatureSensor sens = (TemperatureSensor)obs;
        System.out.printf("display: %g%n",
            sens.getTemp() );
    }
}
```

```
public class ObserverDemo {
    public static void main( String[] args ) {
        TemperatureSensor ts = new TemperatureSensor();
        ts.addObserver( new NotifyWorld() );
        ts.addObserver( new UpdateDisplay() );
        ts.setTemp( 16.0 );
    }
}
```

Decorator

The Decorator pattern is used to add new behaviours to an object, without inheritance. Inheritance is used for adding new behaviours but must be specified at compile-time. Decorator allows new behaviours to be added at run-time!

Decorator

The Decorator pattern is used to add new behaviours to an object, without inheritance. Inheritance is used for adding new behaviours but must be specified at compile-time. Decorator allows new behaviours to be added at run-time!

e.g. In the Java API the basic class for reading character streams is Reader. Reader is abstract but has the following concrete subclasses:

Decorator

The Decorator pattern is used to add new behaviours to an object, without inheritance. Inheritance is used for adding new behaviours but must be specified at compile-time. Decorator allows new behaviours to be added at run-time!

e.g. In the Java API the basic class for reading character streams is Reader. Reader is abstract but has the following concrete subclasses:

- FileReader: Used to read characters from a file

Decorator

The Decorator pattern is used to add new behaviours to an object, without inheritance. Inheritance is used for adding new behaviours but must be specified at compile-time. Decorator allows new behaviours to be added at run-time!

e.g. In the Java API the basic class for reading character streams is Reader. Reader is abstract but has the following concrete subclasses:

- FileReader: Used to read characters from a file
- BufferedReader: Allows text to be read from a character-input stream in manageable chunks (e.g. lines)

Decorator

The Decorator pattern is used to add new behaviours to an object, without inheritance. Inheritance is used for adding new behaviours but must be specified at compile-time. Decorator allows new behaviours to be added at run-time!

e.g. In the Java API the basic class for reading character streams is Reader. Reader is abstract but has the following concrete subclasses:

- FileReader: Used to read characters from a file
- BufferedReader: Allows text to be read from a character-input stream in manageable chunks (e.g. lines)
- LineNumberReader: Keeps track of line numbers

Decorator

The Decorator pattern is used to add new behaviours to an object, without inheritance. Inheritance is used for adding new behaviours but must be specified at compile-time. Decorator allows new behaviours to be added at run-time!

e.g. In the Java API the basic class for reading character streams is Reader. Reader is abstract but has the following concrete subclasses:

- FileReader: Used to read characters from a file
- BufferedReader: Allows text to be read from a character-input stream in manageable chunks (e.g. lines)
- LineNumberReader: Keeps track of line numbers

Decorator

The Decorator pattern is used to add new behaviours to an object, without inheritance. Inheritance is used for adding new behaviours but must be specified at compile-time. Decorator allows new behaviours to be added at run-time!

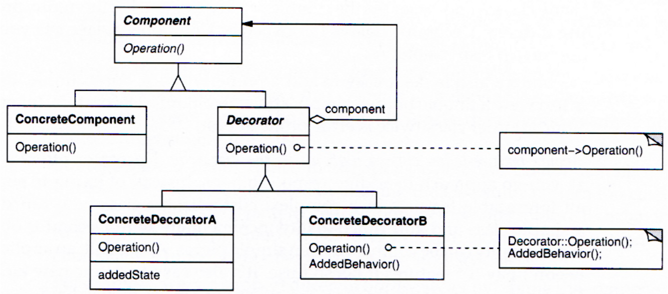
e.g. In the Java API the basic class for reading character streams is Reader. Reader is abstract but has the following concrete subclasses:

- FileReader: Used to read characters from a file
- BufferedReader: Allows text to be read from a character-input stream in manageable chunks (e.g. lines)
- LineNumberReader: Keeps track of line numbers

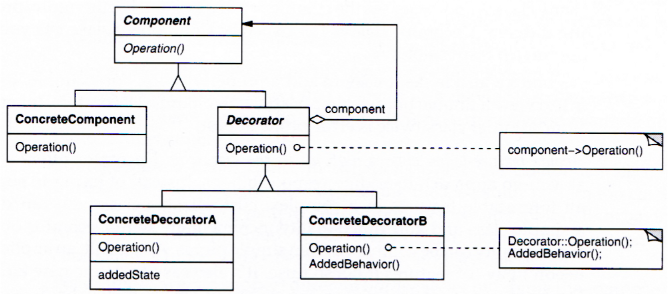
The constructors for BufferedReader and LineNumberReader require a Reader. We build a Reader with the required set of behaviour by instantiating a chain of objects—with each one being decorated by the next.

```java
import java.io.IOException;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.LineNumberReader;

public class BufferLineCountReader {
    public static void main( String[] args ) throws IOException {
        if( args.length != 1 ) {
            System.out.println(
                "usage: java BufferLineCountReader file");
            System.exit( 1 );
        }

        FileReader fr = new FileReader( args[0] );
        BufferedReader br = new BufferedReader( fr );
        LineNumberReader lnr = new LineNumberReader( br );

        String line = null;
        while( (line=lnr.readLine()) != null ) {
            System.out.printf("%5d: %s%n",
                              lnr.getLineNumber(), line );
        }
        lnr.close();
    }
}
```

Here is the general structure of Decorator from [Gamma et al.(1995)Gamma, Helm, Johnson, and Vlissides]:

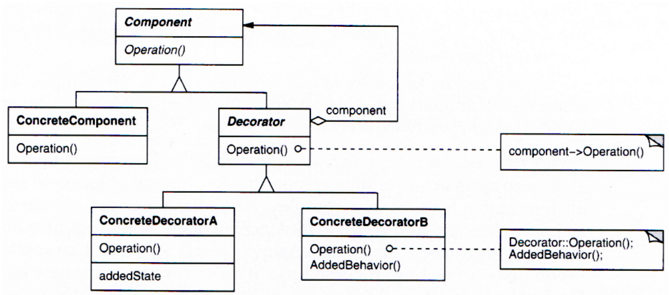


Here is the general structure of Decorator from [Gamma et al.(1995)Gamma, Helm, Johnson, and Vlissides]:



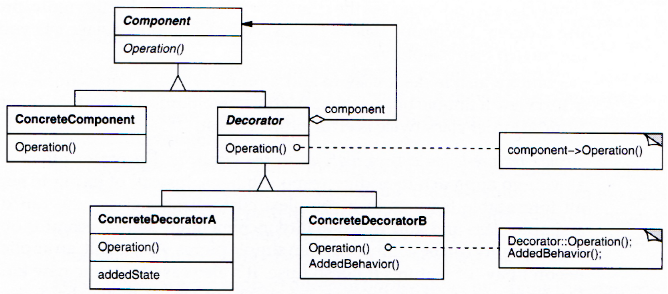
- Component: Defines the interface for objects that can be decorated with new behaviours (e.g. Reader)

Here is the general structure of Decorator from
[Gamma et al.(1995)Gamma, Helm, Johnson, and Vlissides]:



- Component: Defines the interface for objects that can be decorated with new behaviours (e.g. Reader)
- ConcreteComponent: A basic object that can be decorated (but is not a Decorator) (e.g. FileReader)

Here is the general structure of Decorator from [Gamma et al.(1995)Gamma, Helm, Johnson, and Vlissides]:



- Component: Defines the interface for objects that can be decorated with new behaviours (e.g. Reader)
- ConcreteComponent: A basic object that can be decorated (but is not a Decorator) (e.g. FileReader)
- Decorator: Decorates some Component (already decorated or a ConcreteComponent) with new behaviour (e.g. BufferedReader, LineNumberReader)

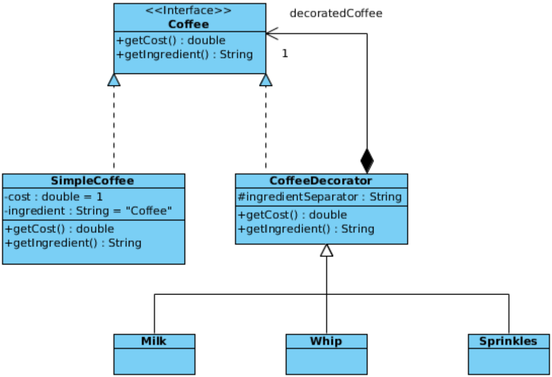
e.g. Coffee

The coffee example from Wikipedia provides a nice introduction to the implementation of Decorator:

`http://en.wikipedia.org/wiki/Decorator_pattern`

e.g. Coffee

The coffee example from Wikipedia provides a nice introduction to the implementation of Decorator:
`http://en.wikipedia.org/wiki/Decorator_pattern`
Here is the class diagram for this example:



Command

The Command pattern treats requests as objects. Instead of a direct function call, we create an object that provides an execute method and stores the parameters of the function call. Instead of simply calling a method of object receiver:

Command

The Command pattern treats requests as objects. Instead of a direct function call, we create an object that provides an execute method and stores the parameters of the function call. Instead of simply calling a method of object receiver:

```
receiver.doStuff(12);
```

Command

The Command pattern treats requests as objects. Instead of a direct function call, we create an object that provides an execute method and stores the parameters of the function call. Instead of simply calling a method of object receiver:

```
receiver.doStuff(12);
```

we create a class to represent such a request. In general, a Command is something with an **execute** method.

Command

The Command pattern treats requests as objects. Instead of a direct function call, we create an object that provides an execute method and stores the parameters of the function call. Instead of simply calling a method of object receiver:

```
receiver.doStuff(12);
```

we create a class to represent such a request. In general, a Command is something with an **execute** method.

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

Command

The Command pattern treats requests as objects. Instead of a direct function call, we create an object that provides an execute method and stores the parameters of the function call. Instead of simply calling a method of object receiver:

```
    receiver.doStuff(12);
```

we create a class to represent such a request. In general, a Command is something with an **execute** method.

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

We need a Command specifically for `receiver.doStuff(int)`...

```java
public class DoStuffCommand implements Command {
    Receiver receiver;
    int value;
    public DoStuffCommand(Receiver receiver, int value) {
        this.receiver = receiver;
        this.value = value;
    }
    public void execute() {
        receiver.doStuff(value);
    }
}
```

```java
public class DoStuffCommand implements Command {
    Receiver receiver;
    int value;
    public DoStuffCommand(Receiver receiver, int value) {
        this.receiver = receiver;
        this.value = value;
    }
    public void execute() {
        receiver.doStuff(value);
    }
}
```

Now lets see how this is used:

```java
public class DoStuffCommand implements Command {
    Receiver receiver;
    int value;
    public DoStuffCommand(Receiver receiver, int value) {
        this.receiver = receiver;
        this.value = value;
    }
    public void execute() {
        receiver.doStuff(value);
    }
}
```

Now lets see how this is used:

```java
import java.util.ArrayList;
public class TestCommand {
    public static void main(String[] args) {
        Receiver receiver = new Receiver();
        ArrayList<Command> commands =
                  new ArrayList<Command>();
                                              //
```

```java
public class DoStuffCommand implements Command {
    Receiver receiver;
    int value;
    public DoStuffCommand(Receiver receiver, int value) {
        this.receiver = receiver;
        this.value = value;
    }
    public void execute() {
        receiver.doStuff(value);
    }
}
```

Now lets see how this is used:

```java
import java.util.ArrayList;
public class TestCommand {
    public static void main(String[] args) {
        Receiver receiver = new Receiver();
        ArrayList<Command> commands =
                     new ArrayList<Command>();
                                                     //
        commands.add( new DoStuffCommand(receiver, 12) );
                                                     //
```

```java
public class DoStuffCommand implements Command {
    Receiver receiver;
    int value;
    public DoStuffCommand(Receiver receiver, int value) {
        this.receiver = receiver;
        this.value = value;
    }
    public void execute() {
        receiver.doStuff(value);
    }
}
```

Now lets see how this is used:

```java
import java.util.ArrayList;
public class TestCommand {
    public static void main(String[] args) {
        Receiver receiver = new Receiver();
        ArrayList<Command> commands =
                  new ArrayList<Command>();
                                                        //
        commands.add( new DoStuffCommand(receiver, 12) );
                                                        //
        // ... other commands added ... time passes ...
                                                        //
```

```java
public class DoStuffCommand implements Command {
    Receiver receiver;
    int value;
    public DoStuffCommand(Receiver receiver, int value) {
        this.receiver = receiver;
        this.value = value;
    }
    public void execute() {
        receiver.doStuff(value);
    }
}
```

Now lets see how this is used:

```java
import java.util.ArrayList;
public class TestCommand {
    public static void main(String[] args) {
        Receiver receiver = new Receiver();
        ArrayList<Command> commands =
                  new ArrayList<Command>();
                                                          //
        commands.add( new DoStuffCommand(receiver, 12) );
                                                          //
        // ... other commands added ... time passes ...
                                                          //
        for (Command cmd : commands)
            cmd.execute();
    }
}
```

This seems like a very indirect way of calling `receiver.doStuff(12)`. But this indirection buys us something. Here are some applications:

This seems like a very indirect way of calling `receiver.doStuff(12)`. But this indirection buys us something. Here are some applications:

- Queue up Commands for later execution

This seems like a very indirect way of calling `receiver.doStuff(12)`. But this indirection buys us something. Here are some applications:

- Queue up Commands for later execution
- Provide logging by modifying `execute` or adding a `log` method

This seems like a very indirect way of calling `receiver.doStuff(12)`. But this indirection buys us something. Here are some applications:

- Queue up Commands for later execution
- Provide logging by modifying `execute` or adding a `log` method
- Support transactions (e.g. bank account transactions) such that new transactions are created not by modifying existing code, but by create new concrete Command classes

This seems like a very indirect way of calling `receiver.doStuff(12)`. But this indirection buys us something. Here are some applications:

- Queue up Commands for later execution
- Provide logging by modifying `execute` or adding a `log` method
- Support transactions (e.g. bank account transactions) such that new transactions are created not by modifying existing code, but by create new concrete Command classes
- Support unlimited undo / redo:

This seems like a very indirect way of calling `receiver.doStuff(12)`. But this indirection buys us something. Here are some applications:

- Queue up Commands for later execution
- Provide logging by modifying `execute` or adding a `log` method
- Support transactions (e.g. bank account transactions) such that new transactions are created not by modifying existing code, but by create new concrete Command classes
- Support unlimited undo / redo:
    - Incorporate an `undo` method into Command to reverse the effects of `execute`

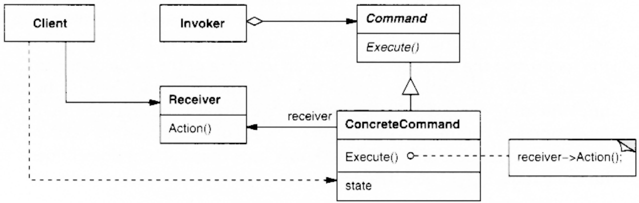
This seems like a very indirect way of calling `receiver.doStuff(12)`. But this indirection buys us something. Here are some applications:

- Queue up Commands for later execution
- Provide logging by modifying `execute` or adding a `log` method
- Support transactions (e.g. bank account transactions) such that new transactions are created not by modifying existing code, but by create new concrete Command classes
- Support unlimited undo / redo:
  - Incorporate an `undo` method into Command to reverse the effects of `execute`
    - Requires some storage of the previous state of the receiver by `execute`

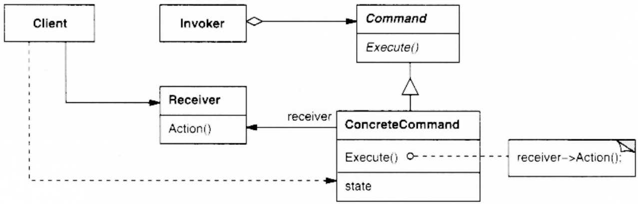
This seems like a very indirect way of calling `receiver.doStuff(12)`. But this indirection buys us something. Here are some applications:

- Queue up Commands for later execution
- Provide logging by modifying `execute` or adding a `log` method
- Support transactions (e.g. bank account transactions) such that new transactions are created not by modifying existing code, but by create new concrete Command classes
- Support unlimited undo / redo:
  - Incorporate an `undo` method into Command to reverse the effects of `execute`
    - Requires some storage of the previous state of the receiver by `execute`
  - Executed commands are maintained in a history list that is traversed backwards for undo operations (by calling `undo`) and forwards for redo operations (by calling `execute`)

The following is the overall structure for Command [Gamma et al.(1995)Gamma, Helm, Johnson, and Vlissides]:



The following is the overall structure for Command [Gamma et al.(1995)Gamma, Helm, Johnson, and Vlissides]:



Sequence diagram:

References

- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.
  Design Patterns: Elements of Reusable Object-Oriented Software.
  Addison-Wesley Professional, 1995.
- Robert C. Martin.
  Agile Software Development: Principles, Patterns, and Practices.
  Prentice Hall, 2003.