A PRIMER ON DESIGN PATTERNS

First Edition

By Rahul Batra

A Primer on Design Patterns

First Edition

Rahul Batra

This book is for sale at http://leanpub.com/aprimerondesignpatterns

This version was published on 2016-03-23

Also By Rahul Batra

A Primer on SQL

A Primer on Java

To Vedant

Contents

Introducing Design Patterns

What are design patterns?

Experience is a great teacher. Suppose you follow this maxim and note down all the great ways you solve common problems that crop up in your day-to-day coding tasks. A decade into your programming life, you would have a neat set of solutions that fit elegantly as solutions to said problems. You have just created a set of **design patterns** for yourself.

It is not only easier, but also more efficient, to stand on the shoulders of giants, so to speak. Programmers have combined and collected design patterns over the decades, ready for you to consume. These solutions to everyday coding problems make your code not only elegant, extensible and readable.

History of design patterns

The first work to explicitly collect and present object oriented design patterns was a book called *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides in 1994. This book and the 23 patterns discussed therein became so popular that they are referred to as the *Gang of Four (GoF) book* and *GoF patterns* respectively.

Pattern Name	Category
Abstract Factory	Creational
Builder	Creational
Factory Method	Creational
Prototype	Creational
Singleton	Creational
Adapter	Structural
Bridge	Structural
Composite	Structural
Decorator	Structural
Facade	Structural
Flyweight	Structural
Proxy	Structural
Chain of Responsibility	Behavioral
Command	Behavioral
Interpreter	Behavioral
Iterator	Behavioral
Mediator	Behavioral

Pattern Name	Category
Memento	Behavioral
Observer	Behavioral
State	Behavioral
Strategy	Behavioral
Template Method	Behavioral
Visitor	Behavioral

Not all GoF patterns gained equal acceptance in the programming community. Some were used more than the others. New patterns have been created since the book was first published. However, quite a few of these patterns remain in widespread use today. We will study a subset of the most important design patterns in the chapters that follow.

The language choice

Since we are going to study object oriented design patterns, it is natural to choose a language that adheres to the OO paradigm. Due to its popularity, I have chosen **Java** as one of the languages to demonstrate the principles of design patterns. However, in some cases, the verbosity of Java leads us away from the core explaination of the pattern and into the innards of language constructs. In such cases, I have decided to remove chunks of boilerplate code which do not illustrate the key points of the pattern being discussed, while still keeping the code Java-like.

The implementation and usefullness of a design pattern varies wildly from language to language. So it would be good to compare the patterns implementation in a contrasting language like **Python** whose dynamic nature and programming philosophy is quite orthogonal to Java.

Pattern abuse

Knowing design patterns is not a substitute for programming taste. If you read this text, or any text on design patterns, as an approach to how *all* code must be designed, you are likely to fall into the pattern abuse habit.

Since the GoF book came out 2 decades ago, there has been much discussion on patterns and they are now accepted as a good thing on balance. But is has also frequently led to over-engineering of software by cramming as many design patterns as one can in the code. You must strive to avoid this trap by using design patterns judiciously and recognising them for what they are - good solutions to common design problems in *certain* but not all scenario's.

Hey, that looks familiar!

If you are an experienced programmer, you would recognize a lot of design patterns as approaches you might have used in the past without formally attaching a name to it. This is a key point about design patterns - they are not arcane knowledge or a magic bullet, they are good solutions. They also are akin to a programming vocabulary, whereby you can describe your approach using a pattern name than by specifying it its entirety.

Decorator Pattern

Consider a *Find* dialog in a text editor. While the basic find operation is pretty simple, there are some options (usually) available to do more advanced searching. We see checkboxes whether or not the search pattern is a regular expression, whether we want to search in the current file or a directory and even an option to enter the replacement text. Let us see if the decorator pattern can help us here.

Permutations and combinations

In our text editor, the `Find` dialog box is a class. When we decide to support regular expression searching, we write a subclass called `FindWithRegEx`.

Listing: the Find class hierarchy

```
1  class FindDialog {
2    public void find (String toFind) {
3      ...
4    }
5    ...
6  }
7
8  class FindWithRegEx extends FindDialog {
9    public void find (String toFind) {
10     ...
11   }
12   ...
13 }
```

We can write another subclass of `FindDialog` which handles the *replace* functionality. And another to search in a directory. But what if we want to have a replace functionality with regular expressions? Do we extend `FindWithRegEx` or the replace class? And what happens when we start dealing with the replace functionality in a directory supporting regular expressions? The permutations and combinations start getting out of control.

Decorating the same entity

The *decorator pattern* is all about keeping the same entity - the *FindDialog* in our case - and decorating it with additional features or responsibilities at runtime. How do we go about keeping the same entity - by implementing the same interface for both the simple find dialog and all it's decorations.

Listing: the top level interface and the simple Find dialog

```
public interface FindDialog {
  public void find();
  public void displayHelp();
}

class SimpleFind implements FindDialog {
  public void find (String toFind) {
    ...
  }

  public void displayHelp() {
    System.out.println("Search current file for text");
  }
}
```

Now we start constructing our decorations. Our goal is to provide such an outline for decorations that it not only adheres to the `FindDialog` interface, it should also provide a barebones structure for further decorator writers. An abstract class which implements the `FindDialog` interface would make a good blueprint for concrete decorators. This class, being abstract, can bind decorator writers to implement certain methods which all decorators should share.

Listing: creating concrete decorators

```
public abstract class FindDecorator implements FindDialog {
  protected FindDialog enhancedFind;

  // Constructor
  public FindDecorator(FindDialog findDialog) {
    this.enhancedFind = findDialog;
  }

  public abstract void find();

  public void displayHelp() {
    enhancedFind.displayHelp();
  }
}

public class FindWithRegEx extends FindDecorator {

  public FindWithRegEx(FindDialog findDialog) {
```

```
19      super(findDialog);
20    }
21
22    public void (String toFind) {
23      ...
24    }
25
26    public void displayHelp() {
27      super.displayHelp();
28      System.out.println("[with Perl compatible regular expressions]");
29    }
```

Using our decorators

We will now create a simple test program to see whether our decoration functionality is working.

Listing: testing our decorator

```
1  public class DecoratorExample {
2    public static void main(String[] args) {
3      FindDialog findDialog = new FindWithRegEx(new SimpleFind());
4      findDialog.displayHelp();
5    }
6  }
```

The interesting line to note here is the construction of the `findDialog` object. This object is a `SimpleFind` decorated with our `FindWithRegEx` decorator. Both the former class and the decorator share the same parent type - the `FindDialog` interface. When the `displayHelp` method is called on this object, the `FindWithRegEx` methos is called which calls its super class method first and then adds its own decoration. The output then is, expectedly, both the lines.

Note that there is no limit to the number of decorations you can apply. Nor are you limited by the order you apply them, because of the super parent interface you created in the beginning.

The decorator stripped to its core

Before we look at the Python implementation of the decorator pattern, let us try to identify the true essence of the decorator pattern. The entire pattern is about providing multiple optional behaviours of a kind of object, and in doing so, the decorated object must adhere to the contract of the original object. Put another way, the caller must not know whether it is dealing with the original object or the decorated one.

Let us now try to build this in Python with minimalism in mind.

Listing: emulating the decorator pattern in Python

```
class SimpleFind:
    def display_help(self):
        print('A simple find dialog')

class FindWithRegEx:
    def __init__(self, findobj):
        self.findobj = findobj

    def display_help(self):
        self.findobj.display_help()
        print('with RegEx support')

f = FindWithRegEx(SimpleFind())
f.display_help()
```

The above code works as expected by we are really playing freely here. Our assumption is that both the classes contain the `display_help` method. What if we wanted to bring some type checking into play here, ensuring at a minimum that both the classes do indeed have such a method?

Enter the Abstract Base Class of Python

We can use the module `abc` to define abstract base classes in Python, which would serve our need to enforce certain contracts by emulating interface-like behavior. Have a look at the code below and notice the `FindDialog` class being defined as an abstract base class.

Listing: using abstract base classes to implement the decorator pattern

```
from abc import ABCMeta, abstractmethod

class FindDialog(metaclass=ABCMeta):
    @abstractmethod
    def display_help(self):
        pass

class SimpleFind(FindDialog):
    def display_help(self):
        print('A simple find dialog')

class FindWithRegEx(FindDialog):
    def __init__(self, findobj):
```

```
        self.findobj = findobj

    def display_help(self):
        self.findobj.display_help()
        print('with RegEx support')

f = FindWithRegEx(SimpleFind())

if not isinstance(f, FindDialog):
    raise TypeError('Our creation is not a find dialog')
else:
    f.display_help()
```

Since we have defined `FindDialog` as an abstract class containing an astract method `display_help()`, we cannot instantiate the class itself. Writing something like below will throw up an error.

```
t = FindDialog()

>Traceback (most recent call last):
  File "Code/python/decorator.py", line 21, in <module>
    t = FindDialog()
TypeError: Can't instantiate abstract class FindDialog with abstract methods dis\
play_help
```

We can see how we have implemented our contract and applied our decorators to the final object `f`. If our decorator was not a subclass of `FindDialog` we would have gotten a runtime `TypeError` which we raised. If our decorator was a subclass of `FindDialog` but did not implement the abstract method `display_help()`, we would have gotten another error.