

Design Patterns in Python

*A learner's approach to understand design patterns
via Python programming language*

Rahul Verma
Chetan Giridhar

Brought to you by: Testing Perspective – www.testingperspective.com

Design Patterns in Python

Copyright © 2011 Rahul Verma, Chetan Giridhar

This book is released under **Creative Commons Attribution-NonCommercial-NoDerivs 3.0 License**, which essentially means that

You are free:

- To share – to copy, distribute and transmit this work

Under the following conditions:

- **Attribution** — You must include a link to www.testingperspective.com along with mentioning the author names – Rahul Verma and Chetan Giridhar.
- **Noncommercial** — You may not use this work for commercial purposes.
- **No Derivative Works** — You may not alter, transform, or build upon this work. You can choose to collaborate with the authors for the next version of this work.

Please check this link for further details:

<http://creativecommons.org/licenses/by-nc-nd/3.0/>

First published: Version 0.1 – February 2011

Published by Testing Perspective

www.testingperspective.com

Cover Image

Renjith Krishnan /FreeDigitalPhotos.net

http://www.freedigitalphotos.net/images/view_photog.php?photogid=721

About the Authors

Rahul Verma

Rahul is a software tester by choice, with focus on technical aspects of the craft. He has explored the areas of security testing, large scale performance engineering and database migration projects. He has expertise in design of test automation frameworks.

Rahul has presented at several conferences, organizations and academic institutions. His articles have been published in various magazines and forums. He runs the website Testing Perspective www.testingperspective.com which he has got the Testing Thought Leadership Award. He is a member of the ISTQB Advanced Level Technical Test Analyst and Foundation Level certification working parties.

You can reach him at rahul_verma@testingperspective.com

Chetan Giridhar

Chetan Giridhar has 5 years experience of working in software services industry, service, product companies and research organization. He runs the website – TechnoBeans technobeans.wordpress.com where he shares his knowledge w.r.t. day-to-day usage of programming for testers.

Chetan has a strong background in C++, Java, Perl and Python. He has developed tools and libraries for users and community developers that could be easily downloaded from [Chetan @ sourceforge](https://sourceforge.net/projects/chetan/). He has written on wide variety of subjects in the field of security, code reviews and agile methodologies for testing magazines including TestingExperience and AgileRecord. He has given lectures on Python Programming at Indian Institute of Astrophysics.

You can reach him at cjgiridhar@gmail.com.

Table of Contents

COPYRIGHT INFORMATION	2
ABOUT THE AUTHORS	3
FOREWARD	6
PREFACE	7
Why Write This Book.....	7
What is a design pattern?.....	7
Context of Design Patterns in Python	8
Design Pattern Classifications	8
Who This Book is For	8
What this book covers.....	8
Pre-Requisites.....	9
Online Version	9
Feedback.....	9
MODEL-VIEW-CONTROLLER PATTERN	10
Controller.....	11
Model.....	11
View	11
A Sample Python Implementation	12
Example Description	12
Database.....	12
Python Code	12
Explanation	14
COMMAND PATTERN	15
A Sample Python Implementation	16
Example description	16
Python Code	16

OBSERVER PATTERN	19
A Sample Python Implementation	20
Example Description	20
Python Code	20
FAÇADE PATTERN	23
A Sample Python Implementation	25
Example Description	25
Python Code	25
MEDIATOR PATTERN	27
A Sample Python Implementation	28
Example Description	28
Python Code	28
FACTORY PATTERN	32
A Sample Python Implementation	33
Example Description	33
Python Code	33
PROXY PATTERN	35
A Sample Python Implementation	36
Example Description	36
Python Code	36
REFERNCES AND FURTHER READING	38

Foreword

Vipul Kocher

[Shri Ramkrishna Paramhans](#), one of the greatest mystics of this age and Guru of [Swami Vivekananda](#), used to narrate a story that went like this...

"There was a Ghost who was very lonely. It is said that when a person dies an accidental death on Tuesday or Saturday becomes a Ghost. Whenever that Ghost saw an accidental death he would rush there hoping he would find a friend. However, every time he was disappointed because all of them went to respective places without even one becoming a Ghost."

While He said it referring to difficulty of finding people with spiritual bent of mind, I take the liberty of using this story for my own interests.

I got drawn to Object Oriented Design world in 1996 despite my job role being that of system tester. I started my career as a developer and that probably was the reason or probably my love for abstract. Whatever be the reason, I got drawn to the world of [Grady Booch](#), [Rumbaugh](#), [Shlaer and Mellor](#), [Coad and Yourdon](#) etc. Then Gang of Four happened. I got blown away by the book [Design Patterns – Elements of Reusable Object-Oriented Software](#). Since then I kept looking for testers who shared same love as me for design patterns. Alas! None was to be found. That is, until I met Rahul Verma and Chetan Giridhar.

I used to wonder when automation framework developers would start talking about the framework as a designer/architect rather than just as a user. Will they ever take automation project as a development project? Will they ever apply design patterns to the automation framework?

In the present book I hope to see this dream of mine fulfilled. While the book aims to talk of design patterns in Python, I hope to see the examples and explanations from the point of view of a tester who wants to create a framework utilizing sound architecture and design patterns.

Patterns have begun to occupy a large portion of my thought process for almost two years now; the process having begun in 1999 with my first attempt at testing patterns by the name [Q-patterns or Questioning-Patterns](#). I sincerely hope to see one piece of the pattern puzzle falling in place with this book.

I hope you enjoy the journey that this book promises to take you through. Vipul Kocher

Vipul Kocher

Co-President, [PureTesting](#)

Preface

"*Testers are poor coders.*" – We here that more often than one would think. There is a prominent shade of truth in this statement as testers mostly code to "get the work done", at times using scripting languages which are vendor specific, just tweaking a recorded script. The quality of code written is rarely a concern and most of the times, the time and effort needed to do good coding is not allocated to test automation efforts.

The above scenario changes drastically when one needs to develop one's own testing framework, extend an existing one or contribute new code to an existing one. To retain the generic property of the framework and to build scalable frameworks, testers need to develop better coding skills.

Why Write This Book

We, the authors of this book, are testers. We are in no way experts in the matter of design patterns or Python. We are learners. We like object oriented programming. We like Python language. We want to be better coders. We have attempted to write this book in very simple language, picking up simple examples to demonstrate a given pattern and being as precise as possible. This is done to provide a text so that the concept of design patterns can reach to those who are not used to formal programming practices. We hope that this would encourage other testers to pick up this subject in the interest better design of test automation.

What is a design pattern?

As per [Wikipedia](#):

A design pattern in architecture and computer science is a formal way of documenting a solution to a design problem in a particular field of expertise. The idea was introduced by the architect Christopher Alexander in the field of architecture and has been adapted for various other disciplines, including computer science. An organized collection of design patterns that relate to a particular field is called a pattern language.

In the object oriented world, design patterns tell us how, in the context of a certain problem, we should structure the classes and objects. They do not translate directly into the solution; rather have to be adapted to suit the problem context.

With knowledge of design patterns, you can talk in "pattern language", because a lot of known design patterns have been documented. In discussions, one could ask,

“Shouldn't we use <design-pattern> for this implementation?” without talking about how classes would be implemented and how objects would be created. It is similar to asking a tester to do Boundary Value Analysis (even BVA should do!), rather than triggering a long talk on the subject.

Design patterns should not be confused with frameworks and libraries. Design patterns are not implementations, though implementations might choose to use them.

Context of Design Patterns in Python

Python is a ground-up object oriented language which enables one to do object oriented programming in a very easy manner. While designing solutions in Python, especially the ones that are more than use-and-throw scripts which are popular in the scripting world, they can be very handy. Python is a rapid application development and prototyping language. So, design patterns can be a very powerful tool in the hands of a Python programmer.

Design Pattern Classifications

- **Creational Patterns** - They describe how best an object can be created. A simple example of such design pattern is a singleton class where only a single instance of a class can be created. This design pattern can be used in situations when we cannot have more than one instances of logger logging application messages in a file.
- **Structural Patterns** – They describe how objects and classes can work together to achieve larger results. A classic example of this would be the façade pattern where a class acts as a façade in front of complex classes and objects to present a simple interface to the client.
- **Behavioral Patterns** – They talk about interaction between objects. Mediator design pattern, where an object of mediator class, mediates the interaction of objects of different classes to get the desired process working, is a classical example of behavioral pattern.

Who This Book is For

If you are a beginner to learning Python or design patterns, this book can prove to be a very easy-to-understand introductory text.

If you are a tester, in addition to the above this book would also be helpful in learning contexts in which design patterns can be used in the test automation world.

What this book covers

This is an initial version of the book covering the following 7 design patterns:

- DP#1 – Model-View-Controller Pattern
- DP#2 – Command Pattern
- DP#3 – Observer Pattern
- DP#4 – Facade Pattern
- DP#5 – Mediator Pattern
- DP#6 – Factory Pattern
- DP#7 – Proxy Pattern

Because the current number of design patterns is a handful, we have not categorized them based on classifications mentioned earlier.

By the time we publish next version of this book with a target of 20 patterns, the content would be organized into classification-wise grouping of chapters.

Pre-Requisites

- Basic Knowledge of Object-Oriented Programming
- Basic Knowledge of Python (see [How Would Pareto Learn Python by Rahul Verma](#) to get started)

Online Version

An online version of the book is available on Testing Perspective website at the following link:

<http://dpip.testingperspective.com/>

This version is a more updated version of this ebook at any given point of time in terms of corrections and updated content. We plan to release updated version of this offline PDF version at regular intervals to keep up with the updated content in online version.


Feedback

There would be mistakes. Some of them would be directly visible and some of them could be more subtle. Please write back to us so that we can correct the same.

You can use the contact form at Testing Perspective website to do so or write to us at feedback@testingperspective.com.

DP#1

The Model-View-Controller Pattern

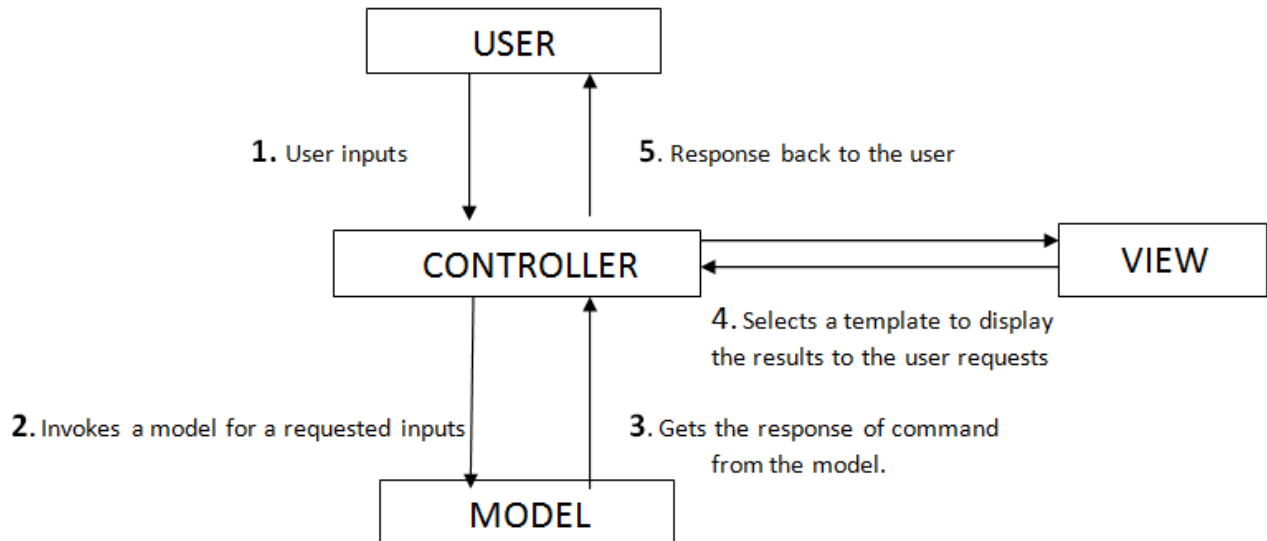
Introduction	
	<p>As per Wikipedia:</p> <p><i>"Model-View-Controller (MVC) is a software architecture, currently considered an architectural pattern used in software engineering. The pattern isolates "domain logic" (the application logic for the user) from input and presentation (GUI), permitting independent development, testing and maintenance of each."</i></p> <p>In MVC Design Pattern, the application is divided into three interacting categories known as the Model, the View and the Controller. The pattern aims at separating out the inputs to the application (the Controller part), the business processing logic (the Model part) and the output format logic (the View part).</p>

In simple words:

- Controller associates the user input to a Model and a View
- Model fetches the data to be presented from persistent storage

- View deals with how the fetched data is presented to the user

A typical diagrammatical representation of the MVC pattern



Let's talk about all these components in detail:

Controller

Controller can be considered as a middle man between user and processing (Model) & formatting (View) logic. It is an entry point for all the user requests or inputs to the application. The controller accepts the user inputs, parses them and decides which type of Model and View should be invoked. Accordingly, it invokes the chosen Model and then the chosen View to provide to the user what he/she/it requested.

Model

Model represents the business processing logic of the application. This component would be an encapsulated version of the application logic. Model is also responsible for working with the databases and performing operations like Insertion, Update and Deletion. Every model is meant to provide a certain kind of data to the controller, when invoked. Further, a single model can return different variants of the same kind of data based on which method of the Model gets called. What exactly gets returned to the controller, could be controlled by passing arguments to a given method of the model.

View

View, also referred as presentation layer, is responsible for displaying the results obtained by the controller from the model component in a way that user wants

them to see or a pre-determined format. The format in which the data can be visible to users can be of any 'type' like HTML or XML. It is responsibility of the controller to choose a view to display data to the user. Type of view could be chosen based on the model chosen, user configuration etc.

A Sample Python Implementation

Example Description

Let's consider a case of a test management system that queries for a list of defects. Here are two typical scenarios that work with any test management system.

- If the user queries for a particular defect, the test management system displays the summary of the defect to the user in a prescribed format.
- If the user searches for a component it shows list of all defects logged against that component.

Database

Let's consider a SQLite DB with the name 'TMS' and a table defects.

TMS.[defects]

ID	Component	Summary
1	XYZ	File doesn't get deleted
2	XYZ	Registry doesn't get created
3	ABC	Wrong title gets displayed

Download the SQLite Database file here: [tms.rar](#)

Python Code

```
# Filename: mvc.py

import sqlite3
import types
class DefectModel:
    def getDefectList(self, component):
        query = "select ID from defects where Component = '%s'" % component
        defectlist = self._dbselect(query)
        list = []
        for row in defectlist:
            list.append(row[0])
```

```
        return list

def getSummary(self, id):
    query = "select summary from defects where ID = '%d' " % id
    summary = self._dbselect(query)
    for row in summary:
        return row[0]
def _dbselect(self, query):
    connection = sqlite3.connect('TMS')
    cursorObj = connection.cursor()
    results = cursorObj.execute(query)
    connection.commit()
    cursorObj.close()
    return results

class DefectView:
    def summary(self, summary, defectid):
        print "#### Defect Summary for defect# %d####\n%s" % (defectid,summary)

    def defectList(self, list, category):
        print "#### Defect List for %s ####\n" % category
        for defect in list:
            print defect

class Controller:
    def __init__(self):
        pass

    def getDefectSummary(self, defectid):
        model = DefectModel()
        view = DefectView()
        summary_data = model.getSummary(defectid)
        return view.summary(summary_data, defectid)

    def getDefectList(self, component):
        model = DefectModel()
        view = DefectView()
        defectlist_data = model.getDefectList(component)
        return view.defectList(defectlist_data, component)
```

Python Code for User

```
import mvc

controller = mvc.Controller()

# Displaying Summary for defect id # 2
```

```
print controller.getDefectSummary(2)

# Displaying defect list for 'ABC' Component
print controller.getDefectList('ABC')
```

Explanation

1. Controller would first get the query from the user. It would know that the query is for viewing defects. Accordingly it would choose DefectModel.
2. If the query is for a particular defect, Controller calls getSummary() method of DefectModel, passing the defect id as the argument, for returning the summary of the defect. Then the Controller calls summary() method of DefectView and displays the response to the user.
3. If the user query consists of a component name, , Controller calls getDefectList() method of DefectModel, passing the component name as the argument, for returning the defect list for the given component. Then the Controller calls defectList() method of DefectView and displays the response to the user.

DP#2

Command Pattern

Introduction



As per [Wikipedia](#):

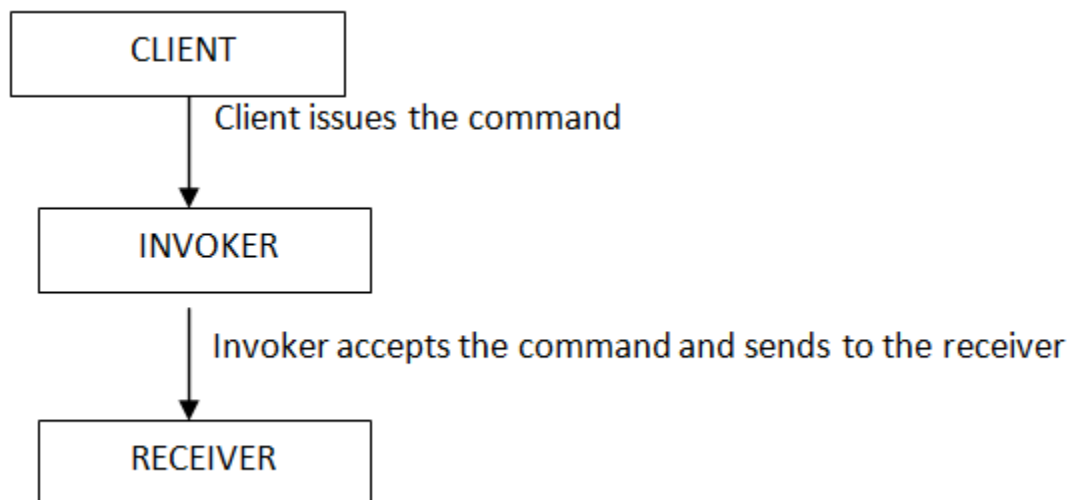
"In object-oriented programming, the command pattern is a design pattern in which an object is used to represent and encapsulate all the information needed to call a method at a later time. This information includes the method name, the object that owns the method and values for the method parameters."

Command pattern is one of the design patterns that are categorized under 'Observer' design pattern. In command pattern the object is encapsulated in the form of a command, i.e., the object contains all the information that is useful to invoke a method anytime user needs. To give an example, let say we have a user interface where in the background of the interface would turn into RED if the button on the user interface named 'RED' is clicked. Now the user is unaware what classes or methods the interface would call to make the background turn RED, but the command user sends (by clicking on 'RED' button) would ensure the background is turned RED. Thus command pattern gives the client (or the user) to use the interface without the information of the actual actions being performed, without affecting the client program.

The key to implementing this pattern is that the **Invoker** object should be kept away from specifics of what exactly happens when its methods are executed. This way, the same **Invoker** object can be used to send commands to objects with similar interfaces.

Command Pattern is associated with three components, the client, the invoker, and the receiver. Let's take a look at all the three components.

- **Client:** the Client represents the one that instantiates the encapsulated object.
- **Invoker:** the invoker is responsible for deciding when the method is to be invoked or called.
- **Receiver:** the receiver is that part of the code that contains the instructions to execute when a corresponding command is given.



A Sample Python Implementation

For demonstrating this pattern, we are going to do a python implementation of an example present on [Wikipedia in C++/Java/C#](#).

Example description

- It's the implementation of a switch
- It could be used to switch on/off
- It shouldn't be hard-coded to switch on/off a particular thing (a lamp or an engine)

Python Code

```
class Switch:
    """ The INVOKER class """

    def __init__(self, flipUpCmd, flipDownCmd):
        self.__flipUpCommand = flipUpCmd
```



```
        self.__flipDownCommand = flipDownCmd

    def flipUp(self):
        self.__flipUpCommand.execute()

    def flipDown(self):
        self.__flipDownCommand.execute()

class Light:
    """The RECEIVER Class"""
    def turnOn(self):
        print "The light is on"

    def turnOff(self):
        print "The light is off"

class Command:
    """The Command Abstract class"""
    def __init__(self):
        pass
        #Make changes

    def execute(self):
        #OVERRIDE
        pass

class FlipUpCommand(Command):
    """The Command class for turning on the light"""

    def __init__(self, light):
        self.__light = light

    def execute(self):
        self.__light.turnOn()

class FlipDownCommand(Command):
    """The Command class for turning off the light"""

    def __init__(self, light):
        Command.__init__(self)
        self.__light = light

    def execute(self):
        self.__light.turnOff()

class LightSwitch:
    """ The Client Class"""
```

```
def __init__(self):
    self.__lamp = Light()
    self.__switchUp = FlipUpCommand(self.__lamp)
    self.__switchDown = FlipDownCommand(self.__lamp)
    self.__switch = Switch(self.__switchUp, self.__switchDown)

def switch(self, cmd):
    cmd = cmd.strip().upper()
    try:
        if cmd == "ON":
            self.__switch.flipUp()
        elif cmd == "OFF":
            self.__switch.flipDown()
        else:
            print "Argument \"ON\" or \"OFF\" is required."
    except Exception, msg:
        print "Exception occurred: %s" % msg

# Execute if this file is run as a script and not imported as a module
if __name__ == "__main__":

    lightSwitch = LightSwitch()

    print "Switch ON test."
    lightSwitch.switch("ON")

    print "Switch OFF test"
    lightSwitch.switch("OFF")

    print "Invalid Command test"
    lightSwitch.switch("****")
```

Observer Pattern

Introduction



As per [Wikipedia](#):

"The observer pattern (a subset of the publish/subscribe pattern) is a software design pattern in which an object, called the subject, maintains a list of its dependants, called observers, and notifies them automatically of any state changes, usually by calling one of their methods. It is mainly used to implement distributed event handling systems."

Typically in the Observer Pattern, we would have:

1. Publisher class that would contain methods for:
 - Registering other objects which would like to receive notifications
 - Notifying any changes that occur in the main object to the registered objects (via registered object's method)
 - Unregistering objects that do not want to receive any further notifications
2. Subscriber Class that would contain:
 - A method that is used by the Publisher Class, to notify the objects registered with it, of any change that occurs.
3. An event that triggers a state change that leads the Publisher to call its notification method

To summarize, Subscriber objects can register and unregister with the Publisher object. So whenever an event, that drives the Publisher's notification method, occurs, the Publisher notifies the Subscriber objects. The notifications would only be passed to the objects that are registered with the Subject at the time of occurrence of the event.

A Sample Python Implementation

Example Description

Let's take the example of a TechForum on which technical posts are published by different users. The users might subscribe to receive notifications when any of the other users publishes a new post. To see this in the light of objects, we could have a 'TechForum' object and we can have another list of objects called 'User' objects that are registered to the 'TechForum' object, that can **observe** for any new posts on the 'TechForum'. Along with the new post notification, the title of the post is sent.

A similar analogy could be drawn in terms of a Job Agency and Job Seekers/Employers. This could be an extension where Employers and Job Seekers subscribe to two different kinds of notification via a common subject (Job Agency). Job Seekers would be looking for relevant job notifications and employers would be looking for new job seeker registration fitting a job profile.

Python Code

```
class Publisher:
    def __init__(self):
        #MAke it uninheritable
        pass

    def register(self):
        #OVERRIDE
        pass

    def unregister(self):
        #OVERRIDE
        pass

    def notifyAll(self):
        #OVERRIDE
        pass

class TechForum(Publisher):
    def __init__(self):
        self._listOfUsers = []
        self.postname = None
```

```
def register(self, userObj):
    if userObj not in self._listOfUsers:
        self._listOfUsers.append(userObj)

def unregister(self, userObj):
    self._listOfUsers.remove(userObj)

def notifyAll(self):
    for objects in self._listOfUsers:
        objects.notify(self.postname)

def writeNewPost(self, postname):
    # User writes a post.
    self.postname = postname
    # When submits the post is published and notification is sent to all
    self.notifyAll()

class Subscriber:
    def __init__(self):
        #make it uninheritable
        pass

    def notify(self):
        #OVERRIDE
        pass

class User1(Subscriber):
    def notify(self, postname):
        print 'User1 notfied of a new post %s' % postname

class User2(Subscriber):
    def notify(self, postname):
        print 'User2 notfied of a new post %s' % postname

class SisterSites(Subscriber):
    def __init__(self):
        self._sisterWebsites = ["Site1", "Site2", "Site3"]
    def notify(self, postname):
        for site in self._sisterWebsites:
            # Send updates by any means
            print "Sent nofication to site: %s" % site

if __name__ == "__main__":
    techForum = TechForum()

    user1 = User1()
    user2 = User2()
```

```
sites = SisterSites()

techForum.register(user1)
techForum.register(user2)
techForum.register(sites)


techForum.writeNewPost("Observer Pattern in Python")

techForum.unregister(user2)

techForum.writeNewPost("MVC Pattern in Python")
```

DP# 4

Facade Pattern

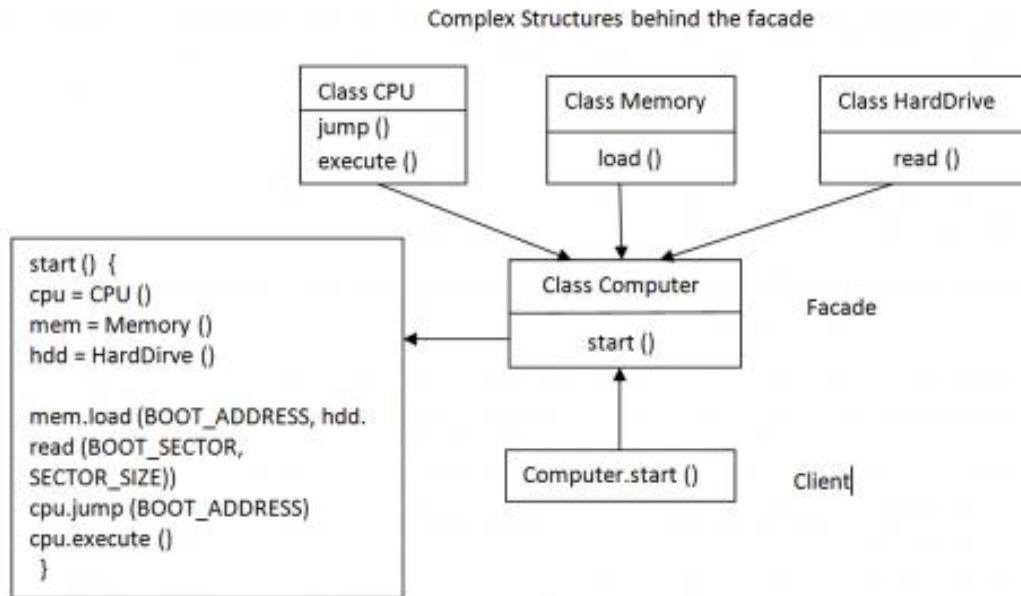
Introduction	
	<p>facade(n): <i>the face or front of a building</i></p> <p>As per Wikipedia:</p> <p><i>"The facade pattern is a software engineering design pattern commonly used with Object-oriented programming. (The name is by analogy to an architectural facade.). A facade is an object that provides a simplified interface to a larger body of code, such as a class library."</i></p>

Facade pattern falls under the hood of Structural Design Patterns. Façade is nothing but an interface that hides the inside details and complexities of a system and provides a simplified "front end" to the client. With façade pattern, client can work with the interface easily and get the job done without being worried of the complex operations being done by the system.

An important point to understand about the Facade pattern is that it provides a simplified interface to a part of the system, thereby providing ease-of-use for a sub-set of the functionality rather than complete functionality. Beauty of this is that the underlying classes are still available to the client if the client wants additional features/greater control and customization that are not provided in the current context of Facade pattern implementation.

Because of the above reason, Facade pattern is not about "encapsulating" the sub-system, rather about providing a simplified interface for a chosen functionality

The pattern can be better explained with a block diagram (based on [Facade pattern example from Wikipedia.](#))



1. In this block diagram, we have three classes representing the CPU, the Memory and the HardDrive of a computer. CPU Class has methods called as `jump()` and `execute()`. Memory Class has a method `load()` and HardDrive Class has `read()` method.
2. We have a Facade, the Class Computer, that is exposed to the Client with the help of the `start()` method.
3. When the Client intends to start the Computer System, it calls the `start()` method of Class Computer by calling `Computer.start()`.

Let's see what actually happens behind the scenes!

In `start()`, CPU, Memory and HardDrive classes are instantiated. Then the `load()` method of Class Memory is called where it gets the `BOOT_ADDRESS` and calls the `read()` method of HardDrive Class from it gets the `BOOT_SECTOR` and `SECTOR_SIZE` of the HardDrive. `start()` then calls the `jump()` method of CPU Class with the `BOOT_ADDRESS` and then calls the `execute()` method.

Thus the client is not aware of the complex operations happening in the background when the computer is started. It only has a facade exposed to it from where it could start the computer easily without bothering about the inner details or complexities.

As discussed earlier, in this implementation, only thing achieved via Facade is “starting” the computer. There could be other finer operations in CPU/Memory/HardDrive classes, which could be achieved by the client only by directly calling their methods.

A Sample Python Implementation

Example Description

Let’s consider the case of a Test Automation Framework. Tests that need to be run for a particular build are written in the form of classes namely, ‘TC1, TC2...TCn’. Each of these classes contains a method called ‘run()’ that gets called to execute the test.

We provide a Facade - TestRunner - on top these Test classes as a simplified interface to execute all tests. In this manner, the client, doesn't need to bother about how many tests really exist and how to execute them. The ‘TestRunner’ class has a method named ‘runAll’ that is responsible for running all the tests that are registered with it.

Now when the user of the automation framework intends to run the tests for a build, as a client, s/he needs to create an object of ‘TestRunner’ class and call the ‘runAll’ method. ‘runAll’ method would inturn create objects of all the Test Classes and call their ‘run’ method., thereby executing all tests.

Python Code

```
#Complex Parts
import time
class TC1:
    def run(self):
        print "##### In Test 1 #####"
        time.sleep(1)
        print "Setting up"
        time.sleep(1)
        print "Running test"
        time.sleep(1)
        print "Tearing down"
        time.sleep(1)
        print "Test Finished\n"

class TC2:
    def run(self):
        print "##### In Test 2 #####"
        time.sleep(1)
        print "Setting up"
        time.sleep(1)
```

```
        print "Running test"
        time.sleep(1)
        print "Tearing down"
        time.sleep(1)
        print "Test Finished\n"

class TC3:
    def run(self):
        print "##### In Test 3 #####"
        time.sleep(1)
        print "Setting up"
        time.sleep(1)
        print "Running test"
        time.sleep(1)
        print "Tearing down"
        time.sleep(1)
        print "Test Finished\n"

#Facade
class TestRunner:
    def __init__(self):
        self.tc1 = TC1()
        self.tc2 = TC2()
        self.tc3 = TC3()

    def runAll(self):
        self.tc1.run()
        self.tc2.run()
        self.tc3.run()

#Client
if __name__ == '__main__':
    testrunner = TestRunner()
    testrunner.runAll()
```

DP#5

Mediator Pattern

Introduction



As per [Wikipedia](#):

" The mediator pattern provides a unified interface to a set of interfaces in a subsystem. This pattern is considered to be a behavioral pattern due to the way it can alter the program's running behavior.."

Typically, mediator pattern is used in cases where many classes start communicating amongst each other to produce result. When the software starts getting developed, more user requests get added and in turn more functionality need to be coded. This results in increased interaction with in the existing classes and in addition of new classes to address new functionality. With the increasing complexity of the system, interaction between classes becomes tedious to handle and maintaining the code becomes difficult.

Mediator pattern comes in as a solution to this problem by allowing loose-coupling between the classes, also called as Colleagues in the Mediator Design Pattern. The idea is that there would be one Mediator class that is aware of the functionality of all the classes in the system. The classes are aware of their functionality and interact with the Mediator class. Whenever there is a need of interaction between the classes, a class sends information to the Mediator and it is the responsibility of the Mediator to pass this information to the required class. Thus the complexity occurring because of interaction between the classes gets reduced.

A Sample Python Implementation

Example Description

A typical example of Mediator Pattern can be manifested in a test automation framework which consists of four classes namely, TC (TestCategory), TestManager, Reporter and DB(Database).

1. Class TC is responsible for running the tests with the help of `setup()`, `execute()` and `tearDown()` methods.
2. Class Reporter calls its `prepare()` method while the test category starts getting executed and calls its `report()` method when the test category finishes its execution. This helps in text based reporting of the tests that are run by the framework.
3. Class DB stores the execution status of the test category by first calling the `insert()` method while the test category is in `setup()`, and then calls the `update()` method once the test category has finished execution. In this way, at any given point of time, the test execution status is available for framework user to query from the database.
4. TestManager Class is the one that co-ordinates for test category execution (Class TC) and fetching the reports (Class Reporter) and getting the test execution status in database (Class DB) with the help of `prepareReporting()` and `publishReport()` methods.
5. Methods `setTM()`, `setTC()`, `setReporter()` and `setDB()` are used so that the classes could register with each other and can communicate easily.

Building the analogy with the Mediator pattern, the TestManager class is a Mediator between Class TC, Class Reporter and Class DB, the Colleagues in the system.

Let's have a look at a sample python code which would make the concept more clear.

Python Code

```
import time
class TC:
    def __init__(self):
        self._tm = tm
        self._bProblem = 0

    def setup(self):
        print "Setting up the Test"
```

```
        time.sleep(1)
        self._tm.prepareReporting()

    def execute(self):
        if not self._bProblem:
            print "Executing the test"
            time.sleep(1)
        else:
            print "Problem in setup. Test not executed."

    def tearDown(self):
        if not self._bProblem:
            print "Tearing down"
            time.sleep(1)
            self._tm.publishReport()
        else:
            print "Test not executed. No tear down required."

    def setTM(self, TM):
        self._tm = tm

    def setProblem(self, value):
        self._bProblem = value

class Reporter:
    def __init__(self):
        self._tm = None

    def prepare(self):
        print "Reporter Class is preparing to report the results"
        time.sleep(1)

    def report(self):
        print "Reporting the results of Test"
        time.sleep(1)

    def setTM(self, TM):
        self._tm = tm

class DB:
    def __init__(self):
        self._tm = None

    def insert(self):
        print "Inserting the execution begin status in the Database"
        time.sleep(1)
```

```
#Following code is to simulate a communication from DB to TC
import random
if random.randrange(1,4) == 3:
    return -1

def update(self):
    print "Updating the test results in Database"
    time.sleep(1)

def setTM(self,TM):
    self._tm = tm

class TestManager:
    def __init__(self):
        self._reporter = None
        self._db = None
        self._tc = None

    def prepareReporting(self):
        rvalue = self._db.insert()
        if rvalue == -1:
            self._tc.setProblem(1)
            self._reporter.prepare()

    def setReporter(self, reporter):
        self._reporter = reporter

    def setDB(self, db):
        self._db = db

    def publishReport(self):
        self._db.update()
        rvalue = self._reporter.report()

    def setTC(self,tc):
        self._tc = tc

if __name__ == '__main__':
    reporter = Reporter()
    db = DB()
    tm = TestManager()
    tm.setReporter(reporter)
    tm.setDB(db)

    reporter.setTM(tm)
    db.setTM(tm)
```

```
# For simplification we are looping on the same test.  
# Practically, it could be about various unique test classes and their objects  
while (1):  
    tc = TC()  
    tc.setTM(tm)  
    tm.setTC(tc)  
    tc.setup()  
    tc.execute()  
    tc.tearDown()
```

- In the above Python Code, the user of the framework first creates instances of Class Reporter, Class DB and Class TestManager and registers these classes with each other with the help of setReporter(), setDB() and setTM() methods.
- When the test category starts execution, the TestManager Class and the TC Class register with each other. TC Class first calls the setup() method which in turn requests the TestManager Class (the Mediator) to be prepared for reporting of the test execution results using the prepareReporting() method.
- prepareReporting() method of the TestManager, the Mediator Class, communicates with the other Colleagues (Class Reporter and Class DB) in the system by calling the prepare() and insert() methods.
- execute() method of Class TC is responsible for running the tests.
- When the test execution is getting finished, tearDown() method of Class TC is called, which in turn calls the publishReport() method of the Mediator (Class TestManager) which communicates with the Colleagues (Class Reporter and Class DB) for preparing the report and getting the execution status in the database (by calling the prepare() and update() methods respectively).
- Also, during insert(), if communication between Class TC (Colleague) and Class TestManager (Mediator) fails, the next step in test execution is conveyed of this failure.

By this example, it can be understood that the communication between Colleagues in the system (Class TC, Class Reporter and Class DB) can easily be achieved by the Mediator (Class TestManager) and the possibility of the complexity that could arise as a result of communication between Colleagues without the Mediator is avoided. This example also demonstrates the capability of a two way interaction between the Colleague and the Mediator with the help of communication failure case between Class TC and Class TestManager.

DP# 6

Factory Pattern

Introduction



As per [Wikipedia](#):

"The factory pattern is a creational design pattern used in software development to encapsulate the processes involved in the creation of objects."

Factory pattern involves creating a super class which provides an abstract interface to create objects of a particular type, but instead of taking a decision on which objects get created it defers this creation decision to its subclasses. To support this there is a creation class hierarchy for the objects which the factory class attempts create and return.

Factory pattern is used in cases when based on a "type" got as an input at run-time, the corresponding object has to be created. In such situations, implementing code based on Factory pattern can result in scalable and maintainable code i.e. to add a new type, one need not modify existing classes; it involves just addition of new subclasses that correspond to this new type.

In short, use Factory pattern when:

- A class does not know what kind of object it must create on a user's request
- You want to build an extensible association between this creator class and classes corresponding to objects that it is supposed to create.

An example would be a better way to understand the above context.

A Sample Python Implementation

Example Description

Let us consider an example to demonstrate the usage of factory pattern (based on [Example of Factory pattern in Java from allapplabs.com.](#))

1. We have a base class called Person that has methods for getting the name and the gender and has two sub-classes namely Male and Female that print the salutation message and a Factory Class.
2. Factory Class has a method named 'getPerson' that takes two arguments 'name' and 'gender' of a person.
3. The Client instantiates the Factory Class and calls the getPerson method with name and gender as arguments.

During run-time, based on the gender, that is passed by the Client, the Factory creates an object of the class pertaining to that gender. Hence the Factory Class at run-time decides which object it needs to create.

Python Code

```
class Person:
    def __init__(self):
        self.name = None
        self.gender = None

    def getName(self):
        return self.name

    def getGender(self):
        return self.gender

class Male(Person):
    def __init__(self, name):
        print "Hello Mr." + name

class Female(Person):
    def __init__(self, name):
        print "Hello Miss." + name

class Factory:
    def getPerson(self, name, gender):
        if gender == 'M':
            return Male(name)
        if gender == 'F':
            return Female(name)
```

```
if __name__ == '__main__':  
    factory = Factory()  
    person = factory.getPerson("Chetan", "M")
```

DP# 7

Proxy Pattern

Introduction



As per [Wikipedia](#):

"A proxy, in its most general form, is a class functioning as an interface to something else. The proxy could interface to anything: a network connection, a large object in memory, a file, or some other resource that is expensive or impossible to duplicate. A well-known example of the proxy pattern is a reference counting pointer object. "

Proxy Pattern is an example of Structural Design Patterns. It is also referred to as "Surrogate" pattern as the intention of this pattern is to create a surrogate for the real object/class.

Proxy pattern has three essential elements:

- Real Subject (that performs the business objectives, represented by Proxy).
- Proxy Class (that acts as an interface to user requests and shields the real Subject)
- Client (that makes the requests for getting the job done).

This design pattern is typically used in situations when:

1. Creation of object for a Real Subject Class is costly in terms of resources and a simple object creation by Proxy Class can be a cheaper option.

2. A need arises that an object must be protected from direct use by its clients.
3. There is a need that an object creation for the Real Subject Class can be delayed to a point when it is actually required.

Some of the real world examples as described by [allapplabs](#) and [userpages](#) of a Proxy Pattern are:

- Website where the Cache Proxy can cache certain set of frequently requested web pages to cater to clients requests. This helps in avoiding many requests getting hit on the server and improves performance.
- The message box which gives the status of a file copy operation giving the progress in terms of percentage completion.
- Opening a large file in a word processor leads to a message saying "Please wait while the software opens the document..."

A Sample Python Implementation

Example Description

Based on the example from [Prof. Kiran](#), let us consider a case of a regular office situation where, in order to speak to a Sales Manager of a company, the Client makes a call first to the receptionist of the sales manager office and then the Receptionist passes the call to the manager. In this case, Sales Manager would be the Subject to whom the Client wants to speak to and the Receptionist would be the Proxy that shields the Subject from talking directly to the Clients.

Expanding this example, we could consider 'Sales Manager' as the Real Subject and create a common Subject Class referred to as 'Managers' from which 'Sales Manager' and the 'Receptionist' can be derived.

Python Code

```
import time
class Manager(object):
    def work(self):
        pass

    def talk(self):
        pass

class SalesManager(Manager):
    def work(self):
        print "Sales Manager working..."
```

```
def talk(self):
    print "Sales Manager ready to talk"

class Proxy(Manager):
    def __init__(self):
        self.busy = 'No'
        self.sales = None

    def work (self):
        print "Proxy checking for Sales Manager availability"
        if self.busy == 'Yes':
            self.sales = SalesManager()
            time.sleep(2);
            self.sales.talk()
        else:
            time.sleep(2);
            print "Sales Manager is busy"

if '__name__' == '__main__':
    p = Proxy()
    p.work()
```

In the above code snippet, the Client has to speak to the Sales Manager. In order to do that it first communicates to the Proxy Class. Proxy, which is the Receptionist in this case, will check if the Subject (in this case the Sales Manager) is busy or not. Depending on the busy status it either passes the call to the Sales Manager or says the Sales Manager is busy. This way the Subject's object creation is delayed until it's actually required and moreover the Subject is shielded from direct usage by the Client.

References and Further Reading

- ***Design Patterns – Elements of Resuable Object-Oriented Software*** – the GoF book (Pearson Publication)
Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides
- ***Head First Design Patterns*** – Oreilly Publication
Eric Freeman and Elisabeth Freeman with Kathey Sierra and Bert Bates
- ***Automated Software Testing*** – Addison-Wesley
Elfriede Dustin, Jeff Rashka and John Paul
- Wikipedia – www.wikipedia.org
- Design Patterns in Python - Alex Martelli (aleax@google.com)
http://www.aleax.it/gdd_pydp.pdf
- Java Design patterns
www.allapplabs.com
- Design Patterns in Python Vespe Savikko
<http://www.python.org/workshops/1997-10/proceedings/savikko.html>
- Object-Oriented Programming in Python
[Michael H Goldwasser](#) (Author), [David Letscher](#) (Author)
- Dr Kiran – IIT Kanpur
<http://www.cse.iitk.ac.in/users/vkirankr/Design%20Patterns%20lecture.ppt>