Learning Python Design Patterns
Second Edition

With the increasing focus on optimized software architecture and design, it is important that software architects think about optimizations in object creation, code structure, and interaction between objects at the architecture or design level. This makes sure that the cost of software maintenance is low and code can be easily reused or is adaptable to change. Building on the success of the previous edition, Learning Python Design Patterns, Second Edition will help you implement real-world scenarios with Python’s latest release, Python v3.5. We start by introducing design patterns from the Python perspective. As you progress through the book, you will learn about Singleton pattern, Factory pattern, and Façade pattern in detail. After this, we’ll look at how to control object access with Proxy pattern. The book also covers Observer pattern, Command pattern, and Compound pattern. By the end of the book, you will have enhanced your professional abilities in software architecture, design, and development.

Who this book is written for
This book is for software architects and Python application developers who are passionate about software design. It will be very useful to engineers with beginner level proficiency in Python and who love to work with Python 3.5.

What you will learn from this book
- Enhance your skills to create a better software architecture
- Understand proven solutions to commonly occurring design issues
- Explore the design principles that form the basis of software design, such as loose coupling, the Hollywood principle, and the open/closed principle, among others
- Delve into object-oriented programming concepts and find out how they are used in software applications
- Develop an understanding of Creational design patterns and the different object creation methods that help you solve issues in software development
- Use Structural design patterns and find out how objects and classes interact to build larger applications
- Focus on the interaction between objects with the command and observer pattern
- Improve the productivity and code base of your application using Python design patterns

Who this book is written for
This book is for software architects and Python application developers who are passionate about software design. It will be very useful to engineers with beginner level proficiency in Python and who love to work with Python 3.5.

What you will learn from this book
- Enhance your skills to create a better software architecture
- Understand proven solutions to commonly occurring design issues
- Explore the design principles that form the basis of software design, such as loose coupling, the Hollywood principle, and the open/closed principle, among others
- Delve into object-oriented programming concepts and find out how they are used in software applications
- Develop an understanding of Creational design patterns and the different object creation methods that help you solve issues in software development
- Use Structural design patterns and find out how objects and classes interact to build larger applications
- Focus on the interaction between objects with the command and observer pattern
- Improve the productivity and code base of your application using Python design patterns

Who this book is written for
This book is for software architects and Python application developers who are passionate about software design. It will be very useful to engineers with beginner level proficiency in Python and who love to work with Python 3.5.

What you will learn from this book
- Enhance your skills to create a better software architecture
- Understand proven solutions to commonly occurring design issues
- Explore the design principles that form the basis of software design, such as loose coupling, the Hollywood principle, and the open/closed principle, among others
- Delve into object-oriented programming concepts and find out how they are used in software applications
- Develop an understanding of Creational design patterns and the different object creation methods that help you solve issues in software development
- Use Structural design patterns and find out how objects and classes interact to build larger applications
- Focus on the interaction between objects with the command and observer pattern
- Improve the productivity and code base of your application using Python design patterns

Who this book is written for
This book is for software architects and Python application developers who are passionate about software design. It will be very useful to engineers with beginner level proficiency in Python and who love to work with Python 3.5.

What you will learn from this book
- Enhance your skills to create a better software architecture
- Understand proven solutions to commonly occurring design issues
- Explore the design principles that form the basis of software design, such as loose coupling, the Hollywood principle, and the open/closed principle, among others
- Delve into object-oriented programming concepts and find out how they are used in software applications
- Develop an understanding of Creational design patterns and the different object creation methods that help you solve issues in software development
- Use Structural design patterns and find out how objects and classes interact to build larger applications
- Focus on the interaction between objects with the command and observer pattern
- Improve the productivity and code base of your application using Python design patterns

Who this book is written for
This book is for software architects and Python application developers who are passionate about software design. It will be very useful to engineers with beginner level proficiency in Python and who love to work with Python 3.5.

What you will learn from this book
- Enhance your skills to create a better software architecture
- Understand proven solutions to commonly occurring design issues
- Explore the design principles that form the basis of software design, such as loose coupling, the Hollywood principle, and the open/closed principle, among others
- Delve into object-oriented programming concepts and find out how they are used in software applications
- Develop an understanding of Creational design patterns and the different object creation methods that help you solve issues in software development
- Use Structural design patterns and find out how objects and classes interact to build larger applications
- Focus on the interaction between objects with the command and observer pattern
- Improve the productivity and code base of your application using Python design patterns

Who this book is written for
This book is for software architects and Python application developers who are passionate about software design. It will be very useful to engineers with beginner level proficiency in Python and who love to work with Python 3.5.

What you will learn from this book
- Enhance your skills to create a better software architecture
- Understand proven solutions to commonly occurring design issues
- Explore the design principles that form the basis of software design, such as loose coupling, the Hollywood principle, and the open/closed principle, among others
- Delve into object-oriented programming concepts and find out how they are used in software applications
- Develop an understanding of Creational design patterns and the different object creation methods that help you solve issues in software development
- Use Structural design patterns and find out how objects and classes interact to build larger applications
- Focus on the interaction between objects with the command and observer pattern
- Improve the productivity and code base of your application using Python design patterns

Who this book is written for
This book is for software architects and Python application developers who are passionate about software design. It will be very useful to engineers with beginner level proficiency in Python and who love to work with Python 3.5.
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 4 'The Façade Pattern – Being Adaptive with Façade'
- A synopsis of the book’s content
- More information on Learning Python Design Patterns - Second Edition
Chetan Giridhar is a technology leader, open source enthusiast, and Python developer. He has written multiple articles on technology and development practices in magazines such as LinuxForYou and Agile Record, and has published technical papers in the Python Papers journal. He has been a speaker at PyCon conferences such as PyCon India, Asia-Pacific, and New Zealand and loves working on real-time communications, distributed systems, and cloud applications. Chetan has been a reviewer at Packt Publishing and has contributed to books on IPython Visualizations and Core Python.
Preface

Design patterns are among the most powerful methods of building large software systems. With an increasing focus on optimized software architecture and design, it is important that software architects think about optimizations in object creation, code structure, and interaction between objects at the architecture or design level. This makes sure that the cost of software maintenance is low, and code can be easily reused and is adaptable to change. Moreover, providing frameworks for reusability and separation of concerns is key to software development today.

What this book covers

Chapter 1, Introduction to Design Patterns, goes through the basics of object-oriented programming and discusses object-oriented design principles in detail. This chapter gives a brief introduction to the concept of design patterns so that you will be able to appreciate the context and application of design patterns in software development.

Chapter 2, The Singleton Design Pattern, covers one of the simplest and well-known Creational design patterns used in application development—the Singleton design pattern. The different ways in which we can create a Singleton pattern in Python are also covered in this chapter along with examples. This chapter also covers the Monostate (or Borg) design pattern, which is a variant of the Singleton design pattern.

Chapter 3, The Factory Pattern – Building Factories to Create Objects, discusses another creational pattern, the Factory pattern. You will also learn about the Factory Method pattern and Abstract Factory pattern with a UML diagram, real-world scenarios, and Python v3.5 implementations.
Chapter 4, The Façade Pattern – Being Adaptive with Façade, shows you another type of design pattern, the Structural design pattern. We will be introduced to the concept of Façade and learn how it is applicable to software design with the help of the Façade design pattern. You'll also learn its implementation with a sample Python application using a real-world scenario.

Chapter 5, The Proxy Pattern – Controlling Object Access, deals with the Proxy pattern that falls into the category of Structural design patterns. We will be introduced to the Proxy as a concept and discuss the design pattern and see how it is used in software application development. You'll also learn about the different variants of the Proxy pattern—Virtual Proxy, Smart Proxy, Remote Proxy, and Protective Proxy.

Chapter 6, The Observer Pattern – Keeping Objects in the Know, talks about the third type of design pattern—the behavioral design pattern. We will be introduced to the Observer design pattern with examples. In this chapter, you'll learn how to implement the Push and Pull models of the Observer pattern and the principles of loose coupling. We'll also see how this pattern is critical when it comes to applying it to cloud applications and distributed systems.

Chapter 7, The Command Pattern – Encapsulating Invocation, tells you about the Command design pattern. We will be introduced to the Command design pattern and discuss how it is used in software application development with a real-world scenario and Python implementation. We will also study two main aspects of the Command pattern—an implementation of redo/rollback operations and asynchronous task execution.

Chapter 8, The Template Method Pattern – Encapsulating Algorithm, discusses the Template design pattern. Like the Command pattern, the Template pattern falls into the category of Behavioral patterns. Here, we discuss the Template method pattern, and you will learn about Hooks with an implementation. We'll also cover the Hollywood principle that helps us appreciate this pattern better.

Chapter 9, Model-View-Controller – Compound Patterns, talks about Compound patterns. We will be introduced to the Model-View-Controller design pattern and discuss how it is used in software application development. MVC is easily one of the most used design patterns; in fact, many Python frameworks are based on this principle. You will learn about the details of MVC implementation with an example application written in Python Tornado (a framework used by Facebook).

Chapter 10, The State Design Pattern, introduces you to the State design pattern, which falls into the category of Behavioral patterns just like the Command or Template design patterns. We will discuss how it is used in software application development.

Chapter 11, AntiPatterns, tells you about AntiPatterns—what we shouldn't do as architects or software engineers.
The Façade Pattern – Being Adaptive with Façade

In the previous chapter, you learned about the Factory design pattern. We discussed about three variations—Simple Factory, Factory method, and Abstract Factory pattern. You also learned how each of them is used in the real world and looked at Python implementations. We also compared the Factory method with Abstract Factory patterns and listed the pros and cons. As we are now aware, both the Factory design pattern and Singleton design pattern (Chapter 2, The Singleton Design Pattern) are classified as Creational design patterns.

In this chapter, we will move ahead and learn about another type of design pattern, the Structural design pattern. We will get introduced to the Façade design pattern and how it is used in software application development. We will work with a sample use case and implement it in Python v3.5.

In brief, we will cover the following topics in this chapter:

- An introduction to Structural design patterns
- An understanding of the Façade design pattern with a UML diagram
- A real-world use case with the Python v3.5 code implementation
- The Façade pattern and principle of least knowledge
Understanding Structural design patterns

The following points will help us understand more about Structural patterns:

- Structural patterns describe how objects and classes can be combined to form larger structures.
- Structural patterns can be thought of as design patterns that ease the design by identifying simpler ways to realize or demonstrate relationships between entities. Entities mean objects or classes in the object-oriented world.
- While the Class patterns describe abstraction with the help of inheritance and provide a more useful program interface, Object patterns describe how objects can be associated and composed to form larger objects. Structural patterns are a combination of Class and Object patterns.

The following are a few examples of different Structural design patterns. You'd notice how each of these involve interaction between objects or classes to achieve high-level design or architectural goals.

Some of the examples of Structural design patterns are as follows:

- **Adapter pattern**: Adapting an interface to another one so that it meets the client's expectations. It tries to match interfaces of different classes based on the client's needs.
- **Bridge pattern**: This decouples an object's interface from its implementation so that both can work independently.
- **Decorator pattern**: This defines additional responsibilities for an object at runtime or dynamically. We add certain attributes to objects with an interface.

There are a few more Structural patterns that you will learn in this book. So, let's start by first taking up the Façade design pattern.

Understanding the Façade design pattern

The façade is generally referred to as the face of the building, especially an attractive one. It can be also referred to as a behavior or appearance that gives a false idea of someone's true feelings or situation. When people walk past a façade, they can appreciate the exterior face but aren't aware of the complexities of the structure within. This is how a façade pattern is used. Façade hides the complexities of the internal system and provides an interface to the client that can access the system in a very simplified way.
Consider the example of a storekeeper. Now, when you, as a customer, visit a store to buy certain items, you're not aware of the layout of the store. You typically approach the storekeeper, who is well aware of the store system. Based on your requirements, the storekeeper picks up items and hands them over to you. Isn't this easy? The customer need not know how the store looks and s/he gets the stuff done through a simple interface, the storekeeper.

The Façade design pattern essentially does the following:

- It provides a unified interface to a set of interfaces in a subsystem and defines a high-level interface that helps the client use the subsystem in an easy way.
- Façade discusses representing a complex subsystem with a single interface object. It doesn't encapsulate the subsystem but actually combines the underlying subsystems.
- It promotes the decoupling of the implementation with multiple clients.

### A UML class diagram

We will now discuss the Façade pattern with the help of the following UML diagram:
As we observe the UML diagram, you'll realize that there are three main participants in this pattern:

- **Façade**: The main responsibility of a façade is to wrap up a complex group of subsystems so that it can provide a pleasing look to the outside world.
- **System**: This represents a set of varied subsystems that make the whole system compound and difficult to view or work with.
- **Client**: The client interacts with the Façade so that it can easily communicate with the subsystem and get the work completed. It doesn't have to bother about the complex nature of the system.

You will now learn a little more about the three main participants from the data structure's perspective.

**Façade**
The following points will give us a better idea of Façade:

- It is an interface that knows which subsystems are responsible for a request
- It delegates the client's requests to the appropriate subsystem objects using composition

For example, if the client is looking for some work to be accomplished, it need not have to go to individual subsystems but can simply contact the interface (Façade) that gets the work done.

**System**
In the Façade world, System is an entity that performs the following:

- It implements subsystem functionality and is represented by a class. Ideally, a System is represented by a group of classes that are responsible for different operations.
- It handles the work assigned by the Façade object but has no knowledge of the façade and keeps no reference to it.

For instance, when the client requests the Façade for a certain service, Façade chooses the right subsystem that delivers the service based on the type of service.
Client

Here's how we can describe the client:

- The client is a class that instantiates the Façade
- It makes requests to the Façade to get the work done from the subsystems

Implementing the Façade pattern in the real world

To demonstrate the applications of the Façade pattern, let's take an example that we'd have experienced in our lifetime.

Consider that you have a marriage in your family and you are in charge of all the arrangements. Whoa! That's a tough job on your hands. You have to book a hotel or place for marriage, talk to a caterer for food arrangements, organize a florist for all the decorations, and finally handle the musical arrangements expected for the event.

In yesteryears, you'd have done all this by yourself, for example by talking to the relevant folks, coordinating with them, negotiating on the pricing, but now life is simpler. You go and talk to an event manager who handles this for you. S/he will make sure that they talk to the individual service providers and get the best deal for you.

Putting it in the Façade pattern perspective:

- **Client**: It's you who need all the marriage preparations to be completed in time before the wedding. They should be top class and guests should love the celebrations.
- **Façade**: The event manager who's responsible for talking to all the folks that need to work on specific arrangements such as food, and flower decorations, among others
- **Subsystems**: They represent the systems that provide services such as catering, hotel management, and flower decorations

Let's develop an application in Python v3.5 and implement this use case. We start with the client first. It's you! Remember, you're the one who has been given the responsibility to make sure that the marriage preparations are done and the event goes fine!
Let's now move ahead and talk about the Façade class. As discussed earlier, the Façade class simplifies the interface for the client. In this case, EventManager acts as a façade and simplifies the work for you. Façade talks to the subsystems and does all the booking and preparations for the marriage on your behalf. Here is the Python code for the EventManager class:

```python
class EventManager(object):
    def __init__(self):
        print("Event Manager:: Let me talk to the folks\n")

    def arrange(self):
        self.hotelier = Hotelier()
        self.hotelier.bookHotel()

        self.florist = Florist()
        self.florist.setFlowerRequirements()

        self.caterer = Caterer()
        self.caterer.setCuisine()

        self.musician = Musician()
        self.musician.setMusicType()
```

Now that we're done with the Façade and client, let's dive into the subsystems. We have developed the following classes for this scenario:

- **Hotelier** is for the hotel bookings. It has a method to check whether the hotel is free on that day (`.isAvailable`).
- **Florist** class is responsible for flower decorations. Florist has the `setFlowerRequirements()` method to be used to set the expectations on the kind of flowers needed for the marriage decoration.
- **Caterer** class is used to deal with the caterer and is responsible for the food arrangements. Caterer exposes the `setCuisine()` method to accept the type of cuisine to be served at the marriage.
- **Musician** class is designed for musical arrangements at the marriage. It uses the `setMusicType()` method to understand the music requirements for the event.
Let us now look at the Hotelier object, followed by Florist object and their methods:

```python
class Hotelier(object):
    def __init__(self):
        print("Arranging the Hotel for Marriage? --")

    def __isAvailable(self):
        print("Is the Hotel free for the event on given day?")
        return True

    def bookHotel(self):
        if self.__isAvailable():
            print("Registered the Booking

")

class Florist(object):
    def __init__(self):
        print("Flower Decorations for the Event? --")

    def setFlowerRequirements(self):
        print("Carnations, Roses and Lilies would be used for Decorations\n\n")

class Caterer(object):
    def __init__(self):
        print("Food Arrangements for the Event --")

    def setCuisine(self):
        print("Chinese & Continental Cuisine to be served\n\n")

class Musician(object):
    def __init__(self):
        print("Musical Arrangements for the Marriage --")

    def setMusicType(self):
        print("Jazz and Classical will be played\n\n")
```
The Façade Pattern – Being Adaptive with Façade

However, you’re being clever here and passing on the responsibility to the event manager, aren’t you? Let's now look at the You class. In this example, you create an object of the EventManager class so that the manager can work with the relevant folks on marriage preparations while you relax.

```python
class You(object):
    def __init__(self):
        print("You:: Whoa! Marriage Arrangements??!!")
    def askEventManager(self):
        print("You:: Let's Contact the Event Manager\n\n")
        em = EventManager()
        em.arrange()
    def __del__(self):
        print("You:: Thanks to Event Manager, all preparations done! Phew!")

you = You()
you.askEventManager()
```

The output of the preceding code is given here:

```
You:: Whoa! Marriage Arrangements??!!
You:: Let's Contact the Event Manager

EventManager:: Let me talk to the folks

Arranging the Hotel for Marriage? --
Is the Hotel free for the event on given day?
Registered the Booking..

Flower Decorations for the Event? --
Carnations, Roses and Lilies would be used for Decorations

Food Arrangements for the Event --
Chinese & Continental Cuisine to be served

Musical Arrangements for the Marriage --
Jazz and Classical will be played

You:: Thanks to Event Manager, all preparations done! Phew!
```
We can relate to the Facade pattern with the real world scenario, in the following way:

- The EventManager class is the Façade that simplifies the interface for You
- EventManager uses composition to create objects of the subsystems such as Hotelier, Caterer, and others

**The principle of least knowledge**

As you have learned in the initial parts of the chapter, the Façade provides a unified system that makes subsystems easy to use. It also decouples the client from the subsystem of components. The design principle that is employed behind the Façade pattern is the **principle of least knowledge**.

The principle of least knowledge guides us to reduce the interactions between objects to just a few friends that are close enough to you. In real terms, it means the following:

- When designing a system, for every object created, one should look at the number of classes that it interacts with and the way in which the interaction happens.
- Following the principle, make sure that we avoid situations where there are many classes created that are tightly coupled to each other.
- If there are a lot of dependencies between classes, the system becomes hard to maintain. Any changes in one part of the system can lead to unintentional changes to other parts of the system, which means that the system is exposed to regressions and this should be avoided.

**Frequently asked questions**

Q1. What is the Law of Demeter and how is it related to the Factory pattern?

A: The Law of Demeter is a design guideline that talks about the following:

1. Each unit should have only limited knowledge of other units in the system
2. A unit should talk to its friends only
3. A unit should not know about the internal details of the object that it manipulates
The principle of least knowledge and Law of Demeter are the same and both point to the philosophy of loose coupling. The principle of least knowledge fits the use case of the Façade pattern as the name is intuitive and the word principle acts as a guideline, not being strict, and being useful only when needed.

Q2. Can there be multiple Façades for a subsystem?
A: Yes, one could implement more than one façade for a group of subsystem components.

Q3. What are the disadvantages of the principle of least knowledge?
A: A Façade provides a simplified interface for the clients to interact with subsystems. In the spirit of providing a simplified interface, an application can have multiple unnecessary interfaces that add to the complexity of the system and reduce runtime performance.

Q4. Can the client access the subsystems independently?
A: Yes, in fact, the Façade pattern provides simplified interfaces so that the client need not be bothered about the complexity of the subsystems.

Q5. Does the Façade add any functionality of its own?
A: A Façade can add its "thinking" to the subsystems, such as making sure that the order of innovation for subsystems can be decided by the Façade.

Summary
We began the chapter by first understanding the Structural design patterns. You then learned about the Façade design pattern and the context in which it's used. We understood the basis of Façade and how it is effectively used in software architecture. We looked at how Façade design patterns create a simplified interface for clients to use. They simplify the complexity of subsystems so that the client benefits.

The Façade doesn't encapsulate the subsystem, and the client is free to access the subsystems even without going through the Façade. You also learned the pattern with a UML diagram and sample code implementation in Python v3.5. We understood the principle of least knowledge and how its philosophy governs the Façade design patterns.

We also covered a section on FAQs that would help you get more ideas on the pattern and its possible disadvantages. We're now geared up to learn more Structural patterns in the chapters to come.
Where to buy this book

You can buy Learning Python Design Patterns - Second Edition from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.