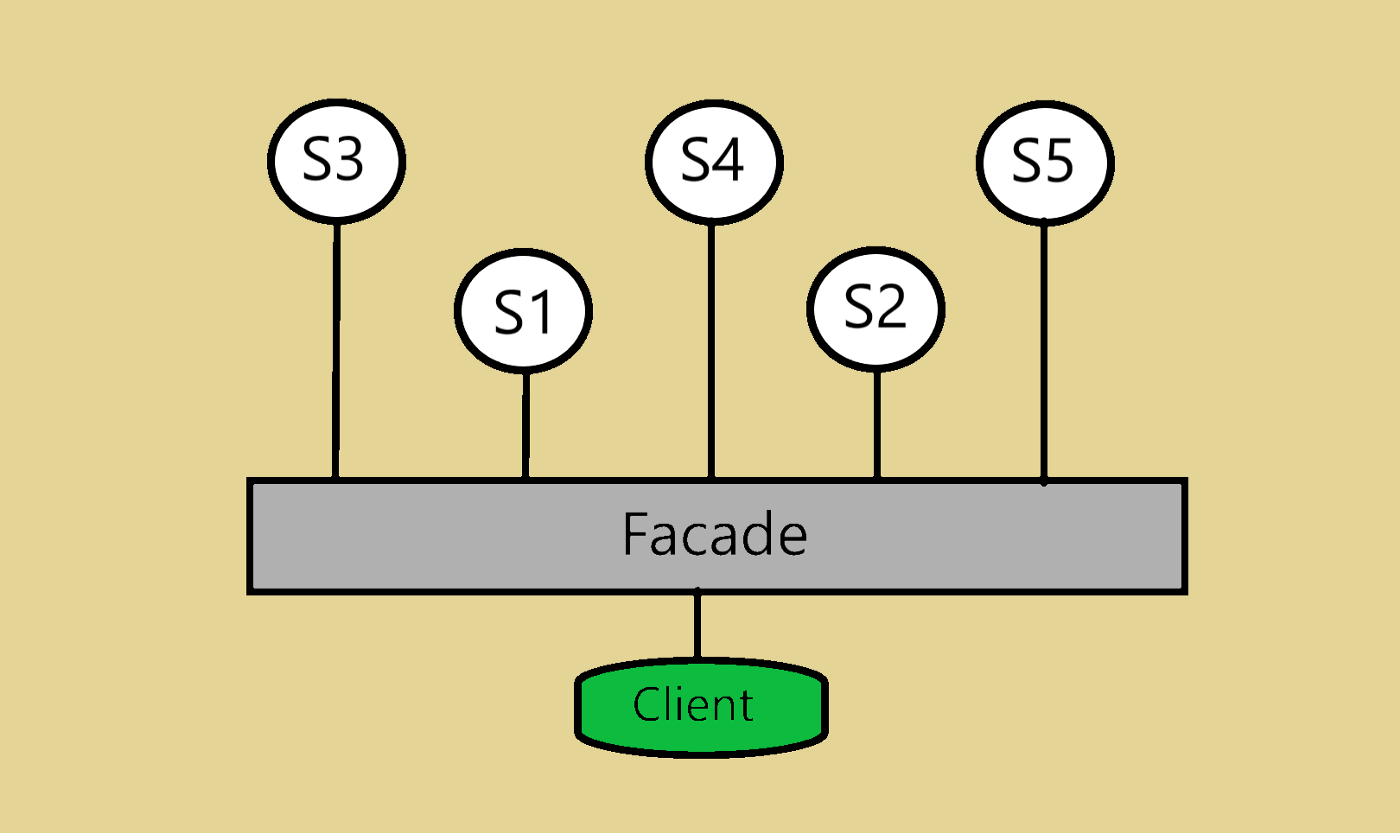
# **Design Patterns in Python – Facade**

Facade pattern allows to hide complexity of system and provide simple interface to client of the service.



Before going through programming example, let’s understand a real life example, Shopkeeper. In shopkeeper, having many items available in the store and he know the exact location of the items in the store but customer doesn’t know all items location. So, we ask shopkeeper with list of items and shopkeeper provides all the items which customer wants to buy as shopkeeper know where items are located. Shopkeeper is Façade of the shop.

Another easy example is facebook-sdk library to user facebook APIs. It provides very easy and simple methods to access all the REST APIs in Pythonic way, so this library is a Façade.

In short, Façade provides interface which can be simple, easy and beautiful to access for client, even though the system may have messy and very much complex subsystem.

There are three points which are involved to understand this pattern.

**Façade Class:** This class is for implementing interface which will be used by client class. This class will use services implemented in system.

**System Class:** Multiple system classes might be there in the system and each system class is for specific purpose.

**Client Class:** Client class is using façade class to access functionality of the system. It could be hard to access system class directly to client is using façade class instead.

Keep following points in your mind before using this pattern to solve your problem in your project or product.

Multiple subsystems should not be heavily dependent on one another.

System is very much complex and hard to access for any client user.

You might want to provide interface for all the functionalities you have.

Let’s understand in programming style with an example of Preparing a food dish.

Preparing food dish includes three processes like Cut the Vegetables, Boil vegetables, Fry Vegetables.

All these three processes are like three subsystem classes. So, lets implement these sub system classes first.

*class Cutter(object):*

*def cutVegetables(self):*

*print(“All Vegetables are cut”)*

*class Boiler(object):*

*def boilVegetable(self):*

*print(“All Vegetables are boiled”)*

*class Frier(object):*

*def fry(self):*

*print(“All vegetables is mixed and fried.”)*

*These above classes are not supposed to be available for client for some reason, So we have implemented façade class for that called COOK.*

*Class Cook(object):*

*def prepareDish(self):*

*Self.cutter = Cutter()*

*Self.cutter.cutVegetables()*

*Self.boiler = Boiler()*

*Self.boiler.boilVegetables()*

*Self.frier = Frier()*

*Self.frier.fry()*

In this class, client need to initiate this class and call prepareDish method to prepare dish and no need to know how all internal processes works. So here Cook is Façade for client.

class Cook(object):  
 def prepareDish(self):  
 self.cutter = Cutter()  
 self.cutter.cutVegetables()  
  
 self.boiler = Boiler()  
 self.boiler.boilVegetables()  
  
 self.frier = Frier()  
 self.frier.fry()  
  
  
class Cutter(object):  
 def cutVegetables(self):  
 print("All vegetables are cut")  
  
  
class Boiler(object):  
 def boilVegetables(self):  
 print("All vegetables are boiled")  
  
  
class Frier(object):  
 def fry(self):  
 print("All vegetables is mixed and fried.")  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 cook = Cook()  
 cook.prepareDish()

Benefits:

Façade class collects all system classes, methods and provide required meaningful APIs so this way for client there will be minimum call. If any changes needs to be done in sub systems, there will not be any change for façade class and that way it will be loosely coupled.

It makes easier to use and maintain bigger structural process.

Point to be noted.

Mediator pattern is very similar to Façade pattern as it also abstracts the functionalities from sub systems similar to Façade. But one notable difference is that sub systems are aware of Mediator class at all.