**Façade Pattern**

**Overview**

Freeman Eric., Freeman Elisa (2004) says that The façade pattern is a unified interface to a set of interfaces in a subsystem in a bigger system. Façade pattern provides higher level interface that makes subsystem easier to use.

In OOP, you use lot of classes to do small/small thing independently. In long term, classes are so many that developer may get lost trying to find useful way to do work. In this heavy weight of different classes, façade pattern can help out restructure code from top level by allowing access to complex structure in easier manner.

Imagine a world with group of objects which interact with each other but you would get lost trying to call methods on each of these objects individually. Well, Façade pattern provides a class/interface that interacts with other classes easily with one method that runs other methods of classes interrelated. It is same as you pressing a button on your remote control and you can play watch tv, and play music through speaker loudly and also play movie through DVD player.

*Façade is outside of house/building. In coding façade provides very outer layer in interface which interacts with inner part of house ( inner part of house) by avoiding complexity. You could do anything inside class to interact with another class which is invoked later but initially you would want to kick off that process in most easy manner. This is called façade pattern in code.*

**Benefits**

* Façade design pattern simplifies complex interaction of objects into simpler firm giving more control and simple way to control interaction.
* A façade pattern decouples client from subsystem, and also simplifies an interface structure.
* It makes client’s interaction simpler and object more reachable without confusion.
* It gives simplified interface that does lot of interaction behind the scene.

**Problem/Scenario**

A customer wants to login into his/her account using username/password. During this process it is crucial to check not only username and password but also validate other flags such as if account is locked or expired if it has subscription. There are so many things under the cover that happens in order to allow user to his/her account.

* Validate username is valid
* Validate password is valid
* Validate user is current user
* Validate user account has not expired
* Validate user account is not locked
* Validate customer answers security question
* Track customer’s location and not allow if it is not in same country where user originally signed up account due to security breach
* Store user’s credential in cookies

**Before Refactor**

Before refactor, it’s very hard to follow the code as lot of objects will be scattered around with maybe circular dependencies where one object is used multiple times and way back and developers would get lost looking at code. Process would look so clumsy and chance that mistake is made or it cant be done properly due to restriction of how objects interact with each other. Also, it would increase codebase complexity a lot if not handled properly.

**After Refactor**

After refactor, a class is created called UserAccountFacade which creates new objects and interacts with methods inside. Processes such as validating username, password, finding if user account is locked or not, validating security question etc. are stored inside subsequent objects. They get invoked by respective object which do initial checks then run the next check based on if previous check is successful. This would have been hard without using this pattern with regular objects.

Before/after link

https://github.com/sppanday/S120-PRT583-Group-A/tree/feat/design-pattern/s260598-PandaySurendra/Sprint-4-Deliverables/Task\_031\_Facade\_Pattern/Task\_031\_Facade\_Pattern

**UML Diagram**

Please find it inside relevant project UML in pattern folder

/FacadePattern.png

**Reference**

Freeman Eric., Freeman Elisa. 2004. 1st ed. O’Reily Media Inc. 1995, Sebasttopo, CA