**ASSIGNMENT # 03**

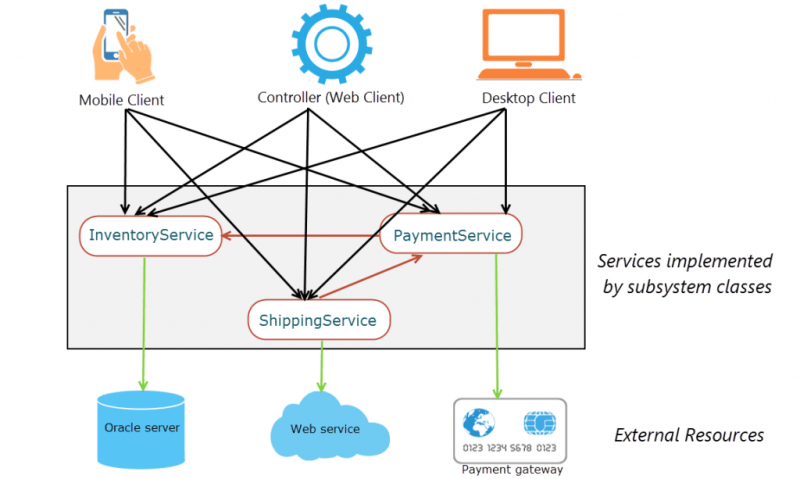
MAJID HUSSAIN

ROLL NUMBER: 0062

**REAL APPLICATIONS OF FACAD PATTERN:**

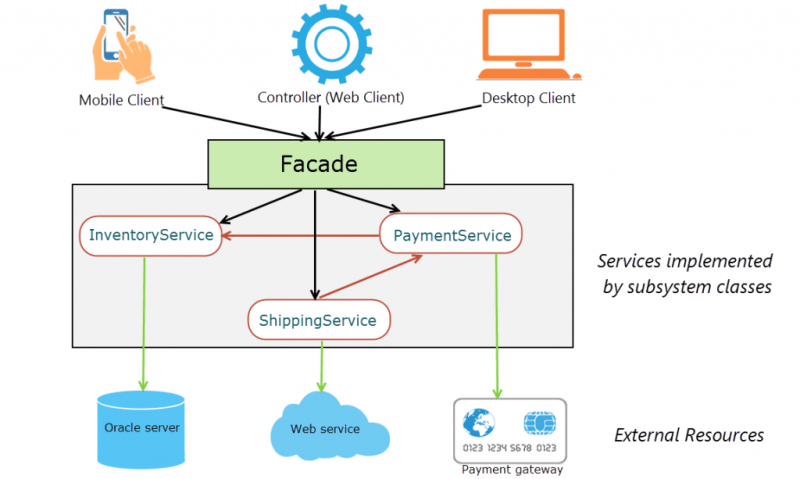
**1. E-commerce mobile client application:**

Our e-commerce store also supports mobile clients. Users can download the client app and place an order from their devices. Legacy desktop clients can also communicate with the store as continuing support for users who wants to place an order over the phone through a customer service assistant. This is how different clients interact with the order fulfillment process of the e-commerce store.



Rather than having the clients tightly coupled to the subsystems, we need is an interface which makes the subsystems easier to use. In our example, our clients just want to place an order. They don’t really need to care about dealing with inventory, shipping or payments. The Facade pattern is a way of providing a simple way for the clients to interact with the subsystems. By working through a facade, now we can make changes to the subsystem classes without affecting the client code. In short we make clients loosely coupled with the subsystem classes.

With a facade, this is how different clients interact with the order fulfillment process.



As you can see in the figure above, with the introduction of a facade, clients now interact with the facade for an order fulfillment instead of individual subsystem services. The facade handles the underlying interactions with the subsystem services transparently from the clients.

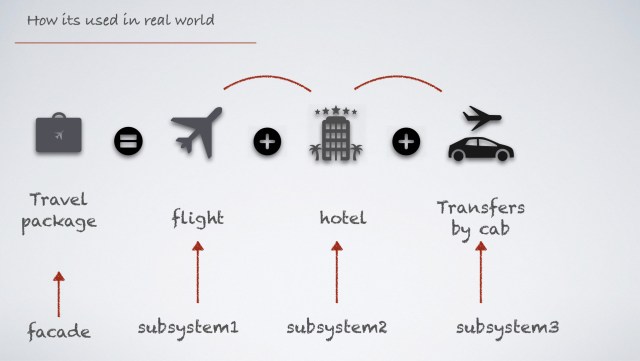
Accordingly, we can categorize the participants of the Facade pattern as:

* **Facade**: Delegates client requests to appropriate subsystem classes.
* **Subsystem classes**: Implements subsystem functionalities. Subsystem classes are used by the facade, but not the other way around. We will come to it later in this post.
* **Client**: Requests the facade to perform some action.

**2. A booking system example:**

Let us consider an example of booking a package. Usually when you try to book a package, the ticket booking system interacts with many of subsystems. The various sub-systems may be flight, hotel and cab booking. In addition this may also interact with many other sub systems.

In this case instead of client having the overhead of interacting with various other subsystems, we can introduce a facade layer which interacts will all these subsystems. Finally once it get the response from all the subsystems, it aggregates all these response and send the response back to the client.



**3. JQUERY APPLICATION OF FACAD:**

The Facade pattern is used quite heavily in jQuery and involves presenting an outward appearance that hides underlying complexity. The idea is that using this pattern, we can provide a simple looking API and obscure the complexity from others. Some examples of the Facade pattern in jQuery are .css (), .animate () and other abstractions like .getJSON(), .get(), and .post(). Take a look at one of the potential implementations of $(document).ready here and you can see just how much complexity is hidden from what appears to be a simple method.

In the facade pattern we abstract quite a bit of detail when the facade function is called. When you find yourself using a library or framework and using many predetermined values (especially for keys in objects), you very well may be using a facade!

**4. For security purposes:**

Use a Facade in two scenarios:

**When I have several external or internal APIs and I want to simplify the interface to access the APIs.**

In the first case I may have a complex API and I use the Facade to effectively hide the real API and expose to the rest of the application only the essential functionalities, the ones that I know are really needed by the application.

By declaring a strict interface by yourself, you can protect the future you and all the other developers in your team from misunderstanding/misuse of the original API.

**5. When I want to protect my code against unstable/on progress API.**

In the second case Facade pattern protect yourself against possible interface changes in the future: whether you are developing a module in a complex application and at the same time other developers are developing other modules, or you are developing an application using an external API in beta phase, there is the possibility that some API you are interacting will evolve and change in the upcoming weeks/months/years. By encapsulating the interactions with the API inside a class of yours, possibly inside an external client library for example (speaking of Java, in a separate jar), when the API eventually will change, you will not need to change your main application, but only the client library that plays the role of Facade. So you will not need to recompile the main application, but just the client library.