"IT IS BETTER TO SEE SOMETHING ONCE THAN TO HEAR ABOUT IT A THOUSAND TIMES."

LET'S EXPLORE PAKISTAN

A Project - A vision to promote the tourism industry in Pakistan

By

Hurmain Javaid	(BSEF19M004)
Nimra Haq	(BSEF19M010)
Iqra Sarwar	(BSEF19M012)

Submitted To

Professor Amna Mirza

DESIGN PATTERNS

Creational:

★ Factory Pattern

“Explore Pakistan” is providing a variety of services to our customers. We will implement a factory of the services due to following reasons:

- ○ If in the future our services expand we will just add a new class implementing a common interface.
- ○ We want to provide service providers the liberty to extend services as they want and the Factory method provides us this flexibility where users can extend internal components.
- ○ We want to save our existing code from breaking when a new service is offered.
- ○ Last but not least, we want to simplify the process for end-users and hide the complex details by offering a generic way to avail of a service.

★ Singleton Pattern

“Explore Pakistan '' has a homePage that contains our vision, introduction, sign-in, and sign-up functionality. We want to apply a singleton pattern to this homepage because of the following reasons:

- ○ This homepage is a common interface and shared resource for all users.
- ○ This page would be accessed by all the instances of the user class (all users.)
- ○ We want to create a global instance of that vision class that would be accessed by all other classes.
- ○ When the application opens, that single shared instance of the homepage is rendered for all users of the app.

★ Prototype Pattern

“Explore Pakistan '' has a hierarchical view that contains collections of tourism spots and services provided. We will apply the prototype pattern to the creation of these collections due to following reasons:

- These collections are stored in a database. We have to access the database and read data every time we want to get a collection.
- Accessing databases and forming these collections is quite an expensive task that can slow down the application considerably.
- By using a prototype we can clone the collections from the existing ones and we don’t need to wait for the collection to be fetched from the database.
- We have applied an observer pattern too. So we don’t need to worry about the updates, they will be synchronized and we will always get the updated collection.

★ Builder Pattern

As mentioned earlier, “Explore Pakistan'' has tourism spots and services. We will apply the builder pattern to their structure due to following reasons:

- Each tourism spot and service object is made up of many elements that are put together to create them.
- For each of them, some properties may be applicable or not. i.e. a hotel may offer discounts and it may be null for others.
- So, to handle variations in building the objects we will use the builder pattern.

Structural:

★ Facade Pattern

“Explore Pakistan'' provides the users the facility to book services and pay for them remotely. We want to apply the facade pattern here because of the following reasons:

- There are a large number of steps and operations involved in booking and payment and we want to wrap them in the simplest possible interface.
- Facade here will hide all the complex details by providing a simple screen to the end-user for booking and payment.
- Payment methods may evolve with time. We want to make the user interface independent of these changes.

★ Composite Pattern

“Explore Pakistan '' has a view that is constructed by a hierarchy of the components. We want to apply the Composite pattern here due to following reasons:

- In the hierarchy of view, we need to treat different things as groups because they exhibit similar properties.
- Similar functionality is associated with all entities of groups. So, by grouping, we can define common functionality.
- This will increase the reusability of the components and functionalities.
- This will make complex views handy and easy to understand.

★ Proxy Pattern

We will be using a proxy pattern in our application when dealing with the database because of the following reasons:

- First and foremost is the security of the app.
- We want to protect the privacy of our users so each access request to the database is validated and then granted.
- There is a single access point to the database.

- In the future, the underlying database structure or other modules may change. Then we will just need to change our single access method. Also, there will be no need to change the way users request access to the database.

Behavioral:

★ Decorator Pattern

"Explore Pakistan '' has two major user classes which are Service Providers and Tourists. We want to apply a decorator pattern to the view for these users in the following manner:

- Views of both user classes contain some functionalities in common. But some features vary depending upon the user class. For example, Both can view Services. but there is variation in the view provided to them.
- We can define a basic view for both user classes with common functionalities and then add the required modification with the help of Decorator Pattern dynamically.

★ Observer Pattern

As mentioned above, "Explore Pakistan'' has a Hierarchical structure of the view. We want to apply the Observer method to the view hierarchy as follows:

- The hierarchy contains tourism spots and services provided at them. Whenever a new service or tourism spot is added, deleted, or existing is modified, the hierarchy needs to update itself.
- So, we will generate an observer method that is responsible for observing the change and update in the hierarchy and syncing the change in view and database, etc.

★ Template Pattern

"Explore Pakistan" is offering the flexibility to customize the UI of the application for which different UI is provided to the users. We want to apply template patterns to the UI modes due to following reasons:

- All of the views have a set of common properties and functionalities that they implement.
- We will create a basic template of the UI which will contain a general blueprint of the UI.
- Then the template pattern is used to create different templates of our basic UI by subclassing.

★ Mediator Pattern

We have a Sign-in and Sign-up form where we are going to use the mediator pattern due to following reasons:

- It uses a loose coupling technique and reduces the risk of higher dependencies by using a mediator element.
- For example, if we apply mediator on our sign-up option, whenever a button is clicked by a user, it will not have to validate the values of all individual form elements so it will not be overburdened.
- Because we will use a mediator element(dialog class). Now its single job is to notify the dialog about the click. Upon receiving this notification, the dialog itself performs the validations or passes the task to the individual elements. Thus, instead of being tied to a dozen form elements, the button is only dependent on the dialog class.