Roll Number: 180101037
Name: Anket Sanjay Kotkar

Introduction to Design Patterns

In software design, Design patterns act as solutions to frequently occurring software problems. They provide a flexible and adaptive way to solve problems in software and recreate them when needed. In designing software, maintenance and extendibility play a significant role. It is always preferable to use design patterns in the form of an outline to solve similar problems to keep the code maintainable and easily understandable for other developers. Using various design patterns helps in this aspect and works as an effective way to work as a team. Design patterns act as a high-level description of solutions and interfaces for general problems. Design patterns generally consist of the following parts: 1) Intent as an introduction to both the problem and its proposed solution. 2) Motivation explains how the solution tackles the problem. 3) Structure describes the internal design pattern structure and how it connected among several internal components. 4) Actual code example supporting the design.

Design patterns are of the following types:

1. **Creational Design Patterns**
2. **Structural Design Patterns**
3. **Behavioural Design Patterns**

Creational Design Patterns:

In **Abstract Factory Pattern**, the patterns aim to create an interface that can create a family or group of objects with a related topic. One nature of the problem that can be solved using this pattern is that we can create objects irrespective of what the platform requirements are as the abstraction takes care to adapt to their variable nature. **Builder Pattern** proposes to use a separate builder type object which will handle all tasks involving different representation and type based class creation in the parent class. This reduces the overall complexity of the code of a very complex system significantly and makes the code better in separately handling events. **Prototype Pattern** uses an already created object to instantiate a new instance of classes, where the new objects are instances of the class we intend to and independent from the class of the object used to create them. Such creation can be seen in Java, an object-oriented language where a base class instantiation can be done using derived class instance. One thing ignored in this type is that some subclasses may be in the structure of the original object structure that is not used and is thus ignored. **Factory Pattern** provides the client with an interface that handles all the queries regarding the creation of objects instances as interfaces leave the task of object creation to the subclasses defined, and the client can construct the objects by calling the factory interface subclass methods instead of directly creating the objects. Apart from this, another pattern of this type is the **Singleton Pattern** which allows instantiating only one class object.

Structural Design Patterns:

It is often possible that two different classes have to interact to exchange the data, but they have incompatible interfaces by their structure. An **Adapter Pattern** is used to resolve this through an adapter connecting the two interfaces, which translates the data taken as input from one class in the format that the other interface can understand and proceed in the task. In **Bridge Pattern**, two classes, namely interface class and implementation class, are decoupled and separately maintained and coded. The interface class contains a call to the class deriving implementation base class. In the Implementation class, the use of encapsulation and class extension is done. **Facade Pattern** is one crucial way to make the code more readable by hiding a group of extensive complex details under the smaller class by calling methods of a large class, which effectively communicates the idea of code parts to the code reviewer. **Proxy Pattern** performs intermediary action between the client and a real-world data or object as accessing the object is not feasible or correct. We can perform intermediate processing on the queries by this pattern and then access the object according to the need.

Behavioural Design Patterns:

In the **Chain of Responsibility Pattern**, there is a sequence of objects connected in a sequence and have well-defined tasks capable of performing. So if the object is unable to perform the task, it sends the query to the next object, and control is now with the next object. **Command Pattern** accepts all the query parameters and turns them into an object instance. In this pattern, this object can query the query further instead of separately handling query parameters. **Iterator Pattern** shows the client all the data items through the interface in a sequential order without the explicit client getting to know the underlying storing or traversing mechanism. The pattern hides all the routing information between the objects. **Memento Pattern** allows the client to undo the changes they did but did not wish to continue and go back. The object's previous state is successfully stored; it cannot be modified but only assigned back to the original value. **Template Pattern** is an adaptive pattern according to client subclasses. The system's parent classes provide abstract methods overridden by the client subclass methods while still keeping the overall algorithm. It uses the features like virtual or abstract methods and method overloading in object-oriented programming languages.

Design patterns in the context of my project:

My project assigned for the Software Engineering Course is **Deal Finder Application**. In the context of my project, the description of design patterns that are useful along with their relevance for the particular part of the project is explained below:

Creational Design Patterns:

The user's wishlist needs to have the product list where every entry corresponds to the product with various attributes. A wishlist is created for every user by default with an empty default list. It should have one to one correspondence with user profiles. So if a user deletes the profile, the wishlist should also get deleted from the database. All this creating, initialization and default behaviour can be combined into a separate builder part to keep the implementation clean effectively using **Builder Pattern**.

Structural Design Patterns:

One part of the code queries various e-commerce websites and fetch all the product list details. Now the data will be extracted (as we thought to implement) by way of scrapping. However, this provides data in HTML format. This data will have to be processed and converted in list format to show the user. For this incompatible type conversion, **Adapter Design Pattern** will be helpful. Further in the product class, the product will have various attributes stored which will be specific to the product along with the methods to go to the parent e-commerce platform providing the product, having the function add to the wishlist on my platform, add the price drop notification alert for the product. **Bridge Pattern** will be helpful for good code implementation and maintainability.

Behavioural Design Pattern:

Chain of Processing Pattern can be used while adding to the wishlist method. In this method, the first handler if the user is logged in. Depending upon the input, the next handler will be either to the login screen or the next handler, which will check if the product is already on the wishlist. If it is, the product will get removed. If it is not, the request to the next handler will add the product to the wishlist. **Iterator pattern** is seen while product list or the offer list is shown to the user on-screen with backend having a completely different structure for showing list keeping in mind to optimize the performance in case of filter applied maximally.