Design Patterns in GO

Creational Patterns

Singleton Pattern:

- ✓ It is a design pattern that lets you ensure that a class has only one instance of an object and no other copies of it.
- ✓ In this pattern one instance is created and the same instance is utilized throughout the program.
- ✓ E.g.: Consider a person, who is a driver in a home where there are 3 cars, the same three cars can be driven by the same driver who is an instance here that is used across the home.

Builder Pattern:

- ✓ This pattern helps us construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.
- ✓ It aims to "Separate the construction of a complex object from its representation so that the same construction process can create different representation".
- ✓ E.g.: Create a housebuilder interface that implements functions like creating basement, walls, terrace, etc.

Factory Pattern:

- ✓ In this pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.
- ✓ We will let the user utilize the required functionalities without exposing the other available functionalities.
- ✓ E.g.: A person tries to access a database to retrieve their info, now we need the database to produce only the data of the person requested and not all the data available within the database.

Abstract Factory Pattern:

- ✓ In this pattern, a new layer of grouping is used to achieve a bigger (and more complex) composite object, which is used through its interfaces.
- ✓ It provides a new layer of encapsulation for factory methods that return common interface for all factories. It groups common factories into a super factory.

- ✓ E.g.: A factory method for vehicle object is created; we will retrieve a vehicle object using a factory returned by the abstract factory. Where vehicle can implement bike, car that implements both interfaces (Vehicle and Car/Bike).

Prototype Pattern:

- ✓ In this pattern, there will be an object or a set of objects that is already created at compilation time, but can be cloned as many times as required at runtime.
- ✓ This is similar to builder pattern but the difference is objects are cloned instead of being built at run-time.
- ✓ E.g.: A shop that sells shirts has Shirt Cloner method to clone the shirts based on required colors and price. Each shirt has a Stock Keeping Unit (SKU) a system to identify items stored at a specific location, that needs to be updated.

Behavioral Patterns

Strategy Pattern:

- ✓ The Strategic Pattern makes use of different algorithms to achieve a specific functionality. All algorithms achieve the same functionality but in a different way.
- ✓ The algorithms are hidden behind an interface and are interchangeable.
- ✓ E.g.: A sort interface with different sorting algorithms, the result achieved is the same but the way each algorithms work is different.

Chain of Responsibility Pattern:

- ✓ The pattern consists of chains, in this case each link of chain follows single responsibility principle (a type/function/method.. must do only one thing and that one thing must be done well).
- ✓ This pattern lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.
- ✓ E.g.: A request to a website to access content, while the content is finished the website runs along other similar content to display for the user until the user exits.

Command Pattern:

- ✓ In this pattern we determine what needs to be done for a particular action irrespective of what the action holds.
- ✓ It passes a request to the request handler without caring to read the request details (like type of request, what it requests,..).

- ✓ E.g.: A box to be delivered in a courier franchise, the delivery partner just transports and delivers the courier without actually knowing the contents within.

Template Pattern:

- ✓ The template patter lets the user write a part of an algorithm while the rest is executed by an abstraction.
- ✓ It is a pattern that defines the skeleton of an algorithm in the super class but let's subclasses override specific steps of the algorithm without changing its structure.
- ✓ E.g.: Consider a user logs in to their account and does a particular task, once the task is done the user might need to re-login to do some other task as the authentication was validated for that session, but this is tiring so instead of logging in for every task the user can login once and the authentication part is shared across sessions as the user performs different tasks. Making sure the authentication part is shared but only the session differs.

Memento Pattern:

- ✓ The Memento pattern saves a finite number of states that are saved from time to time. This helps in recovering back to one of the saved state. It has three parts: Memento, Originator, and Care Taker.
- ✓ It lets you save and restore the previous state of an object without revealing the details of its implementation.
- ✓ E.g.: Consider a string as a state and save the state of the string object, later change the state of the object string and try to recover to the previous state by getting the state from caretaker stack.

Interpreter Pattern:

- ✓ The interpreter pattern helps to create a language to perform some common operations. In simple words SQL is an interpreter language used to perform operations on relational databases.
- ✓ This pattern acts as a connecting path to execute a specific operation.
- ✓ E.g.: A polish notation calculator, a calculator that takes in a mathematical notation and provides an output(Sample: + 3 4 gives 3+4=7)

Visitor Pattern:

- ✓ In this patter, we separate the logic needed to work with a specific object outside of the object itself.
- ✓ This pattern lets you separate algorithms from the objects on which they operate.
- ✓ E.g.: A log appender, it is a function that appends the date, time and sender name with the string to the console.

State Pattern:

- ✓ The State pattern is directly related to FSM (Finite State Machine), it has one or more states and travels between them to execute some behavior.

- ✓ This pattern is very useful with states as with FSM we can achieve very complex behaviors by splitting their scope between states.
- ✓ E.g.: A number guessing game to guess a number in between 1 to 10 in random. Here the state lies between 1 and 10(10 states) and each state is traversed until the right number at that point of time is guessed.

Mediator Pattern:

- ✓ It is a pattern that lies between two types to exchange information to avoid tight coupling.
- ✓ The pattern restricts direct communications between the objects and forces them to collaborate only via a mediator object.
- ✓ E.g.: A hotel where the orders being placed are sent by a ticket to the chef, the ticket has table number and the dish. Here the ticket acts as the mediator between the waiter and the chef.

Observer Pattern:

- ✓ This pattern acts as a trigger for a sequence of actions. When an action occurs it triggers the related subscribed events to perform an action.
- ✓ The pattern lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.
- ✓ E.g.: In YouTube when you subscribe to a channel, whenever an action occurs (new video/ post) an event is triggered that notify us of the action taking place.

Structural Patterns

Composite Pattern:

- ✓ In this pattern, we create compositions ("has a" type of relationship). This will create hierarchies and trees of objects. Objects have different objects with their own fields and methods inside them.
- ✓ It is a structural design pattern that lets you compose objects into tree structures and then work with these structures as if they were individual objects.
- ✓ E.g.: In an organization, there are general managers (Composite) and under general managers, there can be managers (Composite) and under managers there can be developers (Leaf).

Adapter Pattern:

- ✓ In this pattern, we can make use of something that was not built for that specific task at the beginning.
- ✓ This pattern helps to maintain open/closed principle making them more predictable.
- ✓ E.g.: An interface is outdated and it is not possible to replace it easily or faster. Instead a new interface is created to deal with current needs which under the hood use implementations of the older interface.

Bridge Pattern:

- ✓ In this pattern, it decouples an abstraction (an object) from its implementation (the thing that objects does). This helps bring flexibility to the struct that change often. Knowing the inputs and outputs of the code, it allows us to change code without knowing much about it.
- ✓ This lets us split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.
- ✓ E.g.: Two printer classes have two different ways of printing each of them. Totally there are 4 different ways an information can be printed (1 printer prints in 2 ways so 2 printers print in a total of 4 different ways).

Proxy Pattern:

- ✓ In this pattern, an object is wrapped to hide some of its characteristics. It hides an object behind the proxy so the features can be hidden, restricted and so on.
- ✓ It lets us provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.
- ✓ E.g.: A cache memory that stores certain data retrieved previously while accessing database. When a new request occurs the cache memory is queried to find if the information already exists before querying the database.

Facade Pattern:

- ✓ This pattern is similar to proxy, but here instead of wrapping a type or hiding it's features, we group many proxies in a single point such as a file or library . This could be Facade pattern.
- ✓ It acts like a front wall of the building that hides the rooms and corridors on the inside. It provides a simplified interface to a library, a framework, or any other complex set of classes.
- ✓ E.g.: Creating a library to access a specific service from a website that has many services. Like uploading a code to GitHub using Push command using CLI.

Decorator Pattern:

- ✓ This pattern allows you to decorate an already existing type with more functional features without actually touching it. It uses an approach similar to Russian dolls, where a doll is placed within a similar doll but slightly larger.
- ✓ The decorator type implements the same interface of the type it decorates, and stores an instance of that type in it's members. This way you can stack as many decorators as you want.
- ✓ E.g.: Consider a type pizza, where the core is pizza and the ingredients(toppings) are the decorator types.

Flyweight Pattern:

- ✓ This pattern allows sharing the state of a heavy object between many instances of some type.
- ✓ It lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object.
- ✓ E.g.: Accessing Gigabytes of information from a website. Let's say if a person is streaming a movie in Disney and many more wants to do the same, creating a new instance each time a stream has started will occupy a lot of space so instead we make a clone of recently watched amidst people and stream which saves space and avoids new instance creation.
