JSON and Asynchronous JS

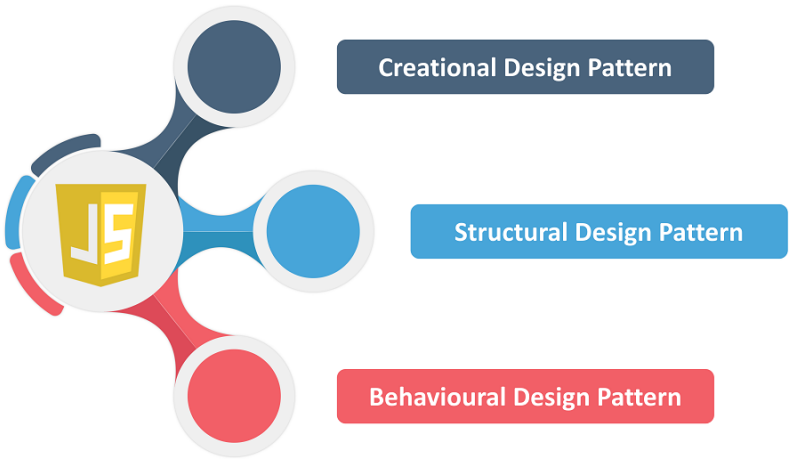
Design Patterns

Design patterns are tried and tested solutions to common problems that arise during software development. They are like recipes that software engineers can follow to build reliable and efficient software. Design patterns are typical solutions to common problems in software design. Each pattern is like a blueprint that you can customize to solve a particular design problem in your code.

For example, imagine you want to make tea. Instead of figuring out the steps and ingredients yourself, you can follow a recipe that has already been tested and proven to work. Similarly, a design pattern in software engineering provides a tested solution to a common problem so that you don't have to start from scratch every time.

Design patterns represent some of the best practices adapted by experienced object-oriented software developers.

There are lots of Design Patterns as they are just an approach of Software Development, we will look at some of the popular patterns :



- Creational patterns: These patterns deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. Creational patterns provide object creation mechanisms that increase flexibility and reuse of existing code.
  eg.: Factory pattern - It provides a way to create objects without specifying the exact class of object that will be created. Imagine you want to make different types of dosa, the factory pattern would be like a recipe that tells you how to make different types of dosas without specifying the ingredients for each type until you decide which dosa you want to make.

- Structural patterns: These patterns deal with object composition, creating relationships between objects to form larger structures. Structural patterns explain how to assemble objects and classes into larger structures, while keeping these structures flexible and efficient.
  eg.: Adapter pattern - It allows two incompatible objects to work together by converting the interface of one object into another. Imagine you have a new phone that uses a different charger than your old phone, the adapter pattern is like an adapter that you can use to connect your new phone to your old charger.
- Behavioral patterns: These patterns deal with communication between objects, what goes on between objects and how they operate together. Behavioral patterns take care of the assignment of responsibilities between objects.
  eg.: Observer pattern - It defines a one-to-many dependency between objects, so that when one object changes state, all its dependents are notified and updated automatically. Imagine you have subscribed to a news website, the observer pattern is

like how the website updates you with the latest news automatically whenever there is a change.

These design patterns provide a way to solve common problems that arise in software development, making the process more efficient, organized and reusable.

Creational Design Patterns

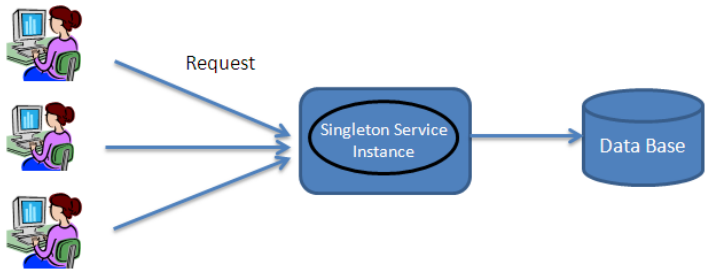
The two most popular design patterns in this class of Patterns are :

1. Singleton Design Patterns
2. Prototype Design Patterns

Singleton Design Pattern

This pattern is used to ensure that a class has only one instance, while providing a global point of access to this instance.

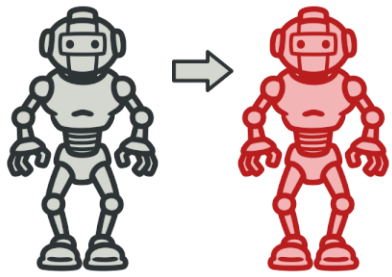
Example: Consider a software system that manages the connections to a database. It is important that there is only one instance of the database connection manager, so that all requests for database connections are handled by the same manager. To achieve this, we can use the singleton pattern to create a DatabaseConnectionManager class that has only one instance. The class can have a static method, such as "getInstance()", that returns the single instance of the class. This way, when any part of the system needs a database connection, it can simply call "DatabaseConnectionManager.getInstance()" to get the single instance of the manager and request a connection.



Prototype Design Pattern

This pattern is used to create new objects by cloning an existing object. In other words we can copy existing objects without making the code dependent on their classes. This is used a lot when we are doing Stress Testing or load testing of the software running on the server.

Example: Consider a software system that needs to create different types of shapes, such as circles, squares, and triangles. Rather than creating new instances of each shape from scratch, we can use the prototype pattern to create a base Shape class that acts as a prototype for the different shapes. Each shape can have a "clone()" method that returns a new instance of itself, created by cloning the original object. This way, when we need a new circle, we can simply call "circlePrototype.clone()" to create a new instance of a circle, without having to recreate it from scratch.



In summary, the Singleton pattern helps to ensure there is only one instance of a class while the Prototype pattern helps to create new objects by cloning existing objects, both making the object creation process more efficient, organized, and reusable.

Do Remember that these are just patterns for software development and in different programming languages it can be implemented differently.

Structural Design Patterns

The two most popular design patterns in this class of Patterns are :

1. Decorator Patterns
2. Proxy Patterns

Decorator Pattern

Decorator is a structural design pattern that can be used to attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors. This pattern is used to dynamically add new functionality to an object, without changing its class.
Example: Consider a Node.js application that needs to send emails. The email sending functionality can be added to a "Mailer" class as a decorator. The decorator can have a method that adds the functionality to send emails with attachments, without changing the class of the Mailer. This way, when we need to send an email with an attachment, we can simply call "attachmentMailer.send()" to send the email with the attachment, while still using the same Mailer class.

Proxy Pattern

Proxy provides a substitute or placeholder for another object. A proxy controls access to the original object, which allows performing something either before or after the request gets through to the original object. This pattern is used to provide a surrogate or placeholder object, which refers to another object.
Example: Consider a Node.js application that needs to access a remote resource, such as an API. Rather than directly accessing the API, we can use the proxy pattern to create a Proxy class that acts as an intermediary between the application and the API. The Proxy class can have a method that makes the API request and returns the response, without the application having to directly access the API. This way, when the application needs to access the API, it can simply call "apiProxy.getData()" to get the data from the API, without having to directly access it.

In summary, the Decorator pattern helps to dynamically add new functionality to an object, while the Proxy pattern helps to provide a surrogate or placeholder object, both making the code structure more flexible and maintainable, even when developing using Node.js.

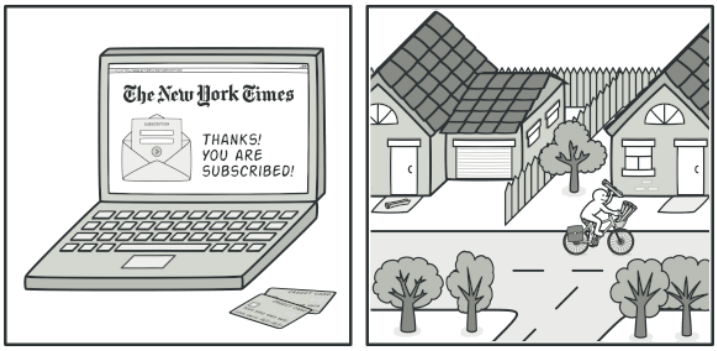
Behavioral Design Patterns

The two most popular design patterns in this class of Patterns are :

1. Observer
2. Memento

Observer

This pattern is used to define a one-to-many relationship between objects, where one object changes and all its dependents are notified and updated automatically. Observer defines a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.
Example: Imagine you are building a chat application and want to notify multiple users when a new message arrives. The Observer pattern can help you to define a one-to-many relationship between objects, where one object changes and all its dependents are notified and updated automatically. In this example, the "ChatRoom" class can be the subject that is being observed, and the "User" class can be the observer that displays the message for a single user. The "ChatRoom" class can have a method to add an observer and notify all the observers when a new message arrives. This way, when a new message is sent, the "ChatRoom" class can notify all the "User" objects, and they can display the message accordingly.



Memento

Memento enables saving and restoring the previous state of an object without revealing the details of its implementation. This pattern is used to capture the current state of an object and store it, so that it can be restored later.
Example: Consider a Node.js application that needs to allow a user to undo and redo actions. The "Editor" class can have a "createMemento()" method to capture the current state of the editor, and a "restore()" method to restore a previously captured state. The "History" class can store all the mementos and provide a way to undo and redo actions. This way, when the user performs an action and wants to undo it, the "Editor" class can restore the previous state using the memento, allowing the user to undo the action.