Design Patterns

Design patterns are typical solutions to commonly occurring problems in software design. They are like pre-made blueprints that you can customize to solve a recurring design problem in your code. You can't just find a pattern and copy it into your program, the way you can with off-the-shelf functions or libraries.

Design patterns are typically categorized into three main groups, based on their purpose:

1. Creational Patterns
2. Structural Patterns
3. Behavioral Patterns

Creational Patterns: These patterns provide various mechanisms to create objects, increasing flexibility and reusability of the code. They control the object instantiation process.

Structural Patterns: These patterns concern the composition of classes and objects. They help ensure that if one part of a system changes, the entire system doesn't have to be altered.

Behavioral Patterns: These patterns are concerned with algorithms and the assignment of responsibilities between objects. They describe how objects communicate with each other.

Singleton Design Pattern

The **Singleton Design Pattern** is used to restrict the instantiation of a class to a single object and provide a global point of access to it. In Node.js, this is particularly useful for managing resources that should be shared across the entire application, such as database connections, configuration settings, or logging services.

Factory Design Pattern

The **Factory Design Pattern** is a creational pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created. In Node.js, it's highly effective for decoupling the client code (the code that needs the object) from the specific concrete classes of the objects being instantiated.

This pattern is especially useful in Node.js for managing different implementations of a service (e.g., logging, data storage, or file parsing) based on configuration or runtime context.

Abstract Factory Design Pattern

The **Abstract Factory Design Pattern** provides an interface for creating families of related or dependent objects without specifying their concrete classes. In Node.js, this pattern is highly effective for managing different application "families" or "kits"—such as multiple database vendors (SQL vs. NoSQL) or various theme configurations (Dark vs. Light UI).

Builder Design Pattern

The **Builder design pattern** is a **creational pattern** that separates the construction of a complex object from its representation. This allows the same construction process to create different representations. It's particularly useful when an object can be created with many different optional parameters or complex construction logic

Prototype Design Pattern

The **Prototype design pattern** is a **creational pattern** used when the object creation process is complex or expensive. It allows you to create new objects by cloning an existing object, known as the prototype, instead of creating a new instance from scratch.

Strategy Design Pattern

- The **Strategy Design Pattern** is a **behavioral pattern** that defines a family of algorithms, encapsulates each one, and makes them interchangeable. It allows the client (or *Context*) to choose the appropriate algorithm (*Strategy*) at runtime without altering the client's code.
- In Node.js (and JavaScript), this pattern is often implemented using classes or plain objects/functions, as JavaScript lacks a native interface concept.

Chain of Responsibility Pattern

- The **Chain of Responsibility** is a **behavioral design pattern** that is often implemented in Node.js, particularly within middleware systems like Express, but it can also be built using pure JavaScript classes for more abstract request handling.
- Here is a detailed breakdown of the pattern and how to implement it in Node.js using classes.

Observer Design Pattern

The Observer design pattern is a behavioral pattern where an object, called the Subject (or Observable), maintains a list of its dependents, called Observers. When the Subject's state changes, it automatically notifies all its Observers, usually by calling one of their methods.

In Node.js, the Observer pattern is fundamentally implemented using the built-in EventEmitter module.

Command Design Pattern

The **Command Design Pattern** is a behavioral pattern that turns a request (an action) into a stand-alone object. This decouples the object that *invokes* the operation (the "sender") from the one that *knows how to perform it* (the "receiver").

In Node.js backend development, this is particularly powerful for creating **task queues**, implementing **undo/redo** functionality (like in text editors or transactional systems), and building extensible **CLI tools**.

Interpreter Design Pattern

The **Interpreter Design Pattern** is a behavioral pattern used to define a grammar for a language and an interpreter to interpret sentences in that language. In simpler terms, it turns a specific syntax (like a math equation or a search query) into an object-oriented structure that can be evaluated.

In Node.js, this pattern is particularly useful when you need to parse custom logic, configuration languages, or mathematical expressions without writing a full-blown compiler.

Iterator Design Pattern

The Iterator Design Pattern is a behavioral pattern that allows you to traverse elements of a collection (like a list, tree, or custom object) without exposing its underlying representation (array, linked list, hash map, etc.).

In Node.js (and modern JavaScript), this pattern is **baked directly into the language** via the **Iteration Protocols** and **Generators**. This makes implementing it much easier and more idiomatic than in classical object-oriented languages like Java or C++.