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++.

Mediator Design Pattern

The Mediator design pattern is a behavioral pattern that acts as a central hub for communication between different objects (components). Instead of objects referring to and calling each other directly (creating a chaotic web of dependencies), they send messages to the Mediator, which then routes those messages to the appropriate recipients.

In Node.js, this is particularly powerful because the ecosystem is heavily based on events. The native EventEmitter class is essentially a built-in implementation of the Mediator pattern.

Memento Design Pattern

- The **Memento Design Pattern** is a behavioral pattern used to save the internal state of an object so that it can be restored later, all without violating encapsulation.
- In the context of an **Express.js** application, this is most commonly used for features like **Undo/Redo functionality**, **version control**, or managing multi-step form wizards where a user might want to step back to a previous state.

Template Method Design Pattern

- The **Template Method Design Pattern** is a behavioral pattern that defines the "skeleton" of an algorithm in a base class but lets subclasses override specific steps of the algorithm without changing its overall structure.
- Think of it like a **cooking recipe**: The steps (boil water, cook pasta, add sauce) are fixed, but the specific *type* of pasta or sauce can be changed by the chef (the subclass).

Visitor Design Pattern

- The **Visitor Design Pattern** is a behavioral pattern that allows you to separate algorithms from the objects on which they operate.
- In Node.js and TypeScript, this is incredibly useful when you have a stable class structure (like a set of product types or a document object model) but you frequently need to add new operations to them (like calculating taxes, generating XML, or validating data) without modifying the classes themselves.

Adapter Design Pattern

- The Adapter Pattern is a structural design pattern that allows objects with incompatible interfaces to collaborate. It acts as a wrapper between two objects. It catches calls for one object and transforms them into a format and interface recognizable by the second object.
- **Real-world Analogy:** Imagine you are traveling from the US to Europe. Your laptop plug (US standard) won't fit into the wall socket (European standard). You need a **power adapter** that accepts your US plug and fits into the European socket.

Bridge Design Pattern

- The Bridge Design Pattern is a structural pattern that lets you 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.

- In TypeScript and Node.js, this is particularly useful when you want to avoid a "Cartesian product" explosion of class subclasses (e.g., avoiding WindowsFileDownloader, LinuxFileDownloader, WindowsFileUploader, LinuxFileUploader).

Composite Design Pattern

- The **Composite Design Pattern** is a structural pattern that lets you compose objects into tree structures to represent part-whole hierarchies. It allows clients to treat individual objects (Leafs) and compositions of objects (Composites) uniformly.
- In the context of **Node.js** and **TypeScript**, this is incredibly useful for scenarios like building file system tools, managing permission trees, organizing UI component structures for server-side rendering, or creating complex task execution pipelines.

Decorator Design Pattern

- The **Decorator Design Pattern** is a structural pattern that allows you to attach new behaviors to objects dynamically by placing these objects inside special wrapper objects that contain the behaviors.
- In TypeScript and Node.js, this pattern is incredibly powerful for adhering to the **Single Responsibility Principle**. Instead of creating a massive class that does everything (logging, caching, validation, and business logic), you create a core class and "decorate" it with extra features.

Facade Design Pattern

- The **Facade Design Pattern** is a structural pattern that provides a simplified, unified interface to a complex set of interfaces within a subsystem.
- Think of it as the "front desk" of a large hotel. Instead of you having to talk to the housekeeping department, the kitchen, and the valet individually, you just speak to the concierge (the Facade), and they coordinate everything behind the scenes for you.

Flyweight Design Pattern

- The Flyweight pattern is a **structural design pattern** focused on optimizing RAM usage. It allows you to fit more objects into the available amount of memory by sharing common parts of the state between multiple objects, rather than keeping all of the data in each object.

Proxy Design Pattern

- The **Proxy Design Pattern** is a structural pattern that provides a substitute or placeholder for another object to control access to it.
- In Node.js and TypeScript, this is incredibly useful for managing resource-intensive operations, adding security layers, implementing caching, or validating data without modifying the original object's logic.