Design Patterns in JavaScript

By. Ahmed Al Ansary

What is Design Pattern?

A **general solution** to **common software design problems**, kind of blueprint or template that you can modify to solve your particular problem

Types of Design Patterns

1. **Creational Patterns**
2. **Structural Patterns**
3. **Behavioral Patterns**
4. Functional Patterns
5. Concurrency Patterns
6. Architectural Patterns
7. ...

https://en.wikipedia.org/wiki/Software_design_pattern

Factory Pattern (Creational)

Providing **object creation mechanisms** that promote **flexibility** and **reusability** of your code especially in situations where you have to create **many different types** of **many different objects**

Factory in a real-world sense: is a building where things are manufactured

Factory in programming sense: is just an object that creates or manufactures different objects

Factory Pattern Example

Let’s say we own **software business** and we have **2 types of employees**

- **developers** who write code for new features
- **testers** who write tests to make sure those features are working

Singleton Pattern (Creational)

When you want to limit the number of instances of an object to at most 1

Singleton Pattern Example

Let’s say we want to write a program that manages a processes and the two main components of the program are

- Process
- Process Manager

We can have one or more processes but only one process manager to manage those processes

Strategy Pattern (Behavioral)

Is a pattern to encapsulate a **group** or a **family** of **closely related algorithms** these algorithms are called **strategies** and all they are just a functions, all the strategy is a function

The strategy pattern comes useful is **it allows you to swap strategies in and out for each other very easily**

Also known as Policy Pattern

Strategy Pattern Example

Let’s say we want to write a program that gives me some **different shipping calculations** for some different companies like FedEx, USPS and UPS

Iterator Pattern (Behavioral)

It allows you to effectively loop over some collection of objects, it’s a common programming task to traverse and manipulate a collection of objects these collections may be stored as arrays or may be something more complex like a tree or graph in addition to being able to traverse these collections you may need to access the items in the collection in a certain order like front to back or back to front, depth-first or breadth-first if you’re dealing with a trees or graphs, or skip every two elements or three elements ..

The iterator pattern allows you to define your own rules of how to traverse some collection of objects

Iterator Pattern Example

We will create an iterator that traverses over an array

Observer Pattern (Behavioral)

Is a design pattern which you define a **one-to-many** dependency relationship from one object known as **the subject** to many other objects known as **the observers** these observers are just a functions which watch the subject and wait for some signal or trigger from the subject before they run kind of reminders or event listeners

It’s very useful when you come to creating event handling systems

Observer Pattern Example

Observer Pattern implementation

Proxy Pattern (Structural)

Allows you to use one **object** known as the proxy as a **placeholder for another object**, proxy can control access to that object so instead of using that object directly, we use the proxy and the proxy uses that object.

Why use a proxy? Is to add an extra functionality

Proxy Pattern Example

Let’s say we’re building a program that displays values for different cryptocurrencies like bitcoin, litecoin or ethereum but in order to get these values we need to make requests to some external api

Mediator Pattern (Behavioral)

Allows you to define an object known as the mediator that encapsulate and controls how some set of objects interact with each other

Mediator Pattern Example

Let’s say we have three objects and these objects need to be able to communicate with each other somehow to share information instead of sending information directly to and from each other, these objects can just send their messages to **one mediator object** and the mediator takes those messages and handles all of the complex logic and routing to decide where these messages need to go

The good example for the mediator pattern is the **chat room**

Visitor Pattern (Behavioral)

Allows you to add or provide a new operations and methods to an object without actually changing that object itself, the new functionality and logic is kept in a **separate object** known as the **visitor object**

Visitor objects are useful when it comes to **extending** some library or framework, the object you want to extend also known as the **receiving object** must have some kind of **accept method** that takes in a visitor object and provides the visitor object **access** to the different information and properties for that object

Visitor Pattern Example

We are going to be creating an employee class the employee will have a name and a salary and we're gonna need to be able to get and set this salary, then we're going to extend the functionality of our employee class using the visitor pattern