Design Patterns in JavaScript: My Top 9

Date: February 6, 2022
Reading Time: 5 minutes
Source: Medium Article

Overview

Design patterns are reusable solutions for common software development problems. Every JavaScript programmer has encountered these issues, and the community has refined solutions over time. These patterns optimize code, enhance readability, and reduce bugs.

Benefits of Design Patterns

- **Proven Solutions:** Tested and optimized by many developers.
- **Reusable:** Generic yet adaptable to specific problems.
- **Expressive:** Elegantly describe complex solutions.
- **Reduce Refactoring:** Lead to cleaner code faster, especially in JavaScript's flexible syntax.
- **Smaller Codebase:** Optimized patterns minimize code, reducing errors.

Brief History of JavaScript

Created by Brendan Eich in 1995 for Netscape, JavaScript started as a "glue" language for HTML. It sparked a browser war with Microsoft's JScript, leading to browser-specific code. ECMAScript standardized the language, with JavaScript as its primary implementation.

Key Features of JavaScript

- **Lightweight and Interpreted:** Uses JIT compilation with a small memory footprint.
- **Object-Oriented with First-Class Functions:** Functions are objects, enabling flexible programming.
- **Flexible:** Supports procedural, object-oriented, and functional styles.
- **Non-Blocking:** Asynchronous operations prevent main thread blocking.

What are Design Patterns?

Design patterns are community-vetted solutions. A **proto-pattern** is a proposed solution needing validation, while an **anti-pattern** (e.g., modifying `Object.prototype`) represents bad practices causing maintenance issues.

Design Pattern Categories

- **Creational:** Optimize object creation.
- **Structural:** Manage object relationships.
- **Behavioral:** Improve object communication.
- **Concurrency:** Handle multi-threading.
- **Architectural:** Define system structure (e.g., MVC).

Top 9 Design Patterns

1. Constructor Pattern

Initializes objects with the `new` keyword, adding properties via dot or bracket notation.

```javascript
function Person(name, age, isDeveloper) {
    this.name = name;
    this.age = age;
    this.isDeveloper = isDeveloper || false;
}
```

```
Person.prototype.writesCode = function() {
    console.log(this.isDeveloper ? "this person writes code" : "this person does not write code");
};
let person1 = new Person("Ana", 32, true);
let person2 = new Person("Bob", 36);
person1.writesCode(); // prints: this person writes code
person2.writesCode(); // prints: this person does not write code
```

2. Module Pattern

Uses closures for encapsulation, separating private and public members.

```
let counterIncrementer = (function() {
    let counter = 0;
    return function() {
        return counter++;
    };
})();
console.log(counterIncrementer()); // prints: 1
console.log(counterIncrementer()); // prints: 2
```

```
let fruitsCollection = (function() {
    let objects = [];
    return {
        addObject: function(object) {
            objects.push(object);
        },
        removeObject: function(object) {
            let index = objects.indexOf(object);
            if (index >= 0) objects.splice(index, 1);
        },
        getObjects: function() {
            return JSON.parse(JSON.stringify(objects));
        }
    };
})();
fruitsCollection.addObject("apple");
fruitsCollection.addObject("orange");
console.log(fruitsCollection.getObjects()); // prints: ["apple", "orange"]
```

Drawbacks: Changing visibility requires updating callers; new methods can’t access private members.

3. Revealing Module Pattern

Exposes private logic via an anonymous object with renamed public methods.

```
let fruitsCollection = (function() {
    let objects = [];
    function addObject(object) {
        objects.push(object);
    }
    function removeObject(object) {
        let index = objects.indexOf(object);
        if (index >= 0) objects.splice(index, 1);
    }
    function getObjects() {
        return JSON.parse(JSON.stringify(objects));
    }
    return {
        addName: addObject,
        removeName: removeObject,
        getNames: getObjects
    };
})();
fruitsCollection.addName("Bob");
fruitsCollection.addName("Alice");
console.log(fruitsCollection.getNames()); // prints: ["Bob", "Alice"]
```

Issues: Fragile with inheritance; private-public mismatches can cause bugs.

4. Singleton Pattern

Ensures a single instance, ideal for configurations.

```
let ConfigurationSingleton = (function() {
    let config;
    function initializeConfiguration(values) {
        this.randomNumber = Math.random();
        values = values || {};
        this.number = values.number || 5;
        this.size = values.size || 10;
    }
    return {
        getConfig: function(values) {
            if (config === undefined) {
                config = new initializeConfiguration(values);
            }
            return config;
        }
    };
})();
let configObject = ConfigurationSingleton.getConfig({ size: 8 });
console.log(configObject); // prints: { number: 5, size: 8, randomNumber: someRandomDecimalValue }
let configObject2 = ConfigurationSingleton.getConfig({ number: 8 });
console.log(configObject2); // prints: { number: 5, size: 8, randomNumber: sameRandomDecimalValue }
```

5. Observer Pattern (Publisher/Subscriber)

Manages event-driven communication with subscribers.

```
let publisherSubscriber = {};
(function(container) {
    let id = 0;
    container.subscribe = function(topic, callback) {
        if (!container[topic]) container[topic] = [];
        let token = { id: id++, callback: callback };
        container[topic].push(token);
        return token;
    };
    container.unsubscribe = function(topic, token) {
        let subscribers = container[topic];
        if (subscribers) {
            container[topic] = subscribers.filter(sub => sub.id !== token.id);
        }
    };
    container.publish = function(topic, data) {
        let subscribers = container[topic];
        if (subscribers) {
            subscribers.forEach(sub => sub.callback(data));
        }
    };
})(publisherSubscriber);
let subscriptionId = publisherSubscriber.subscribe("mouseClicked", function(data) {
    console.log("mouseClicked: " + JSON.stringify(data));
});
publisherSubscriber.publish("mouseClicked", { x: 10, y: 20 }); // prints: mouseClicked: {"x":10,"y":20}
publisherSubscriber.unsubscribe("mouseClicked", subscriptionId);
```

6. Mediator Pattern

Centralizes communication, e.g., for a wizard form.

```
let mediator = (function() {
    let steps = {};
    return {
        register: function(name, step) {
            steps[name] = step;
        },
        next: function(currentStep) {
            if (steps[currentStep].next) {
                steps[currentStep].next();
            }
        }
    };
})();
let step1 = { next: function() { console.log("Step 2"); } };
mediator.register("step1", step1);
mediator.next("step1"); // prints: Step 2
```

Drawback: Single point of failure.

7. Prototype Pattern

Uses prototypes for inheritance and performance.

```
let personPrototype = {
    sayHi: function() {
        console.log("Hi, I am " + this.name + " and I have " + this.age);
    }
};
function Person(name, age) {
    this.name = name;
    this.age = age || 26;
}
Person.prototype = personPrototype;
let person1 = new Person("John Doe");
person1.sayHi(); // prints: Hi, I am John Doe and I have 26
```

8. Command Pattern

Encapsulates calls for API abstraction.

```
let invoker = {
    add: function(x, y) { return x + y; },
    subtract: function(x, y) { return x - y; }
};
let commandManager = {
    execute: function(name, ...args) {
        if (invoker[name]) {
            return invoker[name].apply(invoker, args);
        }
        return false;
    }
};
console.log(commandManager.execute("add", 3, 5)); // prints: 8
console.log(commandManager.execute("subtract", 5, 3)); // prints: 2
```

Drawback: Adds abstraction layer, potential performance hit.

9. Facade Pattern

Simplifies complex interfaces, e.g., jQuery selectors.

```
let facade = {
    select: function(query) {
        // Simplified DOM selection logic
        return document.querySelectorAll(query);
    }
};
console.log(facade.select(".parent .child div")); // returns NodeList
```

Drawback: Minor performance overhead.

Conclusion

Design patterns enhance JavaScript development by improving maintainability and readability. For deeper study, read *Design Patterns: Elements of Reusable Object-Oriented Software* by the Gang of Four or *Learning JavaScript Design Patterns* by Addy Osmani.