JavaScript Design Patterns

The ultimate guide to the most useful design patterns

Soumyajit Pathak
May 14, 2018 · 15 min read ★

UPDATE NOTE: *Updated the Proxy Pattern example to use ES6 Proxy and Reflect. Replaced images of source code snippets with* ***GitHub*** *gists.*

In this article, we are going to talk about design patterns that can be and should be used to write better, maintainable JavaScript code. I assume you have a basic understanding of JavaScript and concepts like classes (classes in JavaScript can be tricky), objects, prototypal inheritance, closures, etc.

This article is a long read as a whole because of the nature of the subject matter, so I have tried to keep the sections self-contained. So you as a reader can pick and choose specific parts (or, in this case, specific patterns) and ignore the ones you are not interested in or are well versed with. Now, let's get started.

> ***Note:*** *Source code for the implementation of all the design patterns explained here is on* ***GitHub****.*

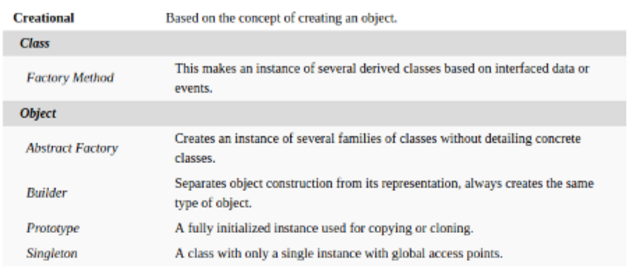
Introduction

We write code to solve problems. These problems usually have many similarities, and, when trying to solve them, we notice several common patterns. This is where design patterns come in.

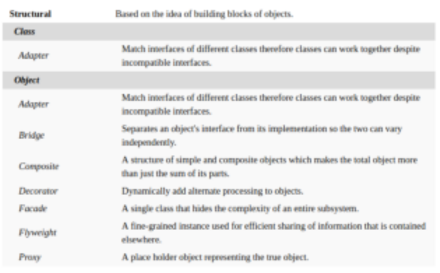
> *A* ***design pattern*** *is a term used in software engineering for a general, reusable solution to a commonly occurring problem in software design.*

The underlying concept of design patterns has been around in the software engineering industry since the very beginning, but they weren't really so formalised. **Design Patterns: Elements Of Reusable Object-Oriented Software** written by **Erich Gamma, Richard Helm, Ralph Johnson**, and **John Vlissides** — the famous Gang of Four (GoF)—was instrumental in pushing the formalised concept of design patterns in software engineering. Now, design patterns are an essential part of software development and have been so for a long time.

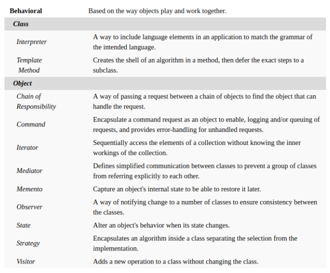
There were 23 design patterns introduced in the original book.



Creational	Based on the concept of creating an object.
Class	
Factory Method	This makes an instance of several derived classes based on interfaced data or events.
Object	
Abstract Factory	Creates an instance of several families of classes without detailing concrete classes.
Builder	Separates object construction from its representation, always creates the same type of object.
Prototype	A fully initialized instance used for copying or cloning.
Singleton	A class with only a single instance with global access points.



Structural	Based on the idea of building blocks of objects.
Class	
Adapter	Match interfaces of different classes therefore classes can work together despite incompatible interfaces.
Object	
Adapter	Match interfaces of different classes therefore classes can work together despite incompatible interfaces.
Bridge	Separates an object's interface from its implementation so the two can vary independently.
Composite	A structure of simple and composite objects which makes the total object more than just the sum of its parts.
Decorator	Dynamically add alternate processing to objects.
Facade	A single class that hides the complexity of an entire subsystem.
Flyweight	A fine-grained instance used for efficient sharing of information that is contained elsewhere.
Proxy	A place holder object representing the true object.



Behavioral	Based on the way objects play and work together.
Class	
Interpreter	A way to include language elements in an application to match the grammar of the intended language.
Template Method	Creates the shell of an algorithm in a method, then defer the exact steps to a subclass.
Object	
Chain of Responsibility	A way of passing a request between a chain of objects to find the object that can handle the request.
Command	Encapsulate a command request as an object to enable, logging and/or queuing of requests, and provides error-handling for unhandled requests.
Iterator	Sequentially access the elements of a collection without knowing the inner workings of the collection.
Mediator	Defines simplified communication between classes to prevent a group of classes from referring explicitly to each other.
Memento	Capture an object's internal state to be able to restore it later.
Observer	A way of notifying change to a number of classes to ensure consistency between the classes.
State	Alter an object's behavior when its state changes.
Strategy	Encapsulates an algorithm inside a class separating the selection from the implementation.
Visitor	Adds a new operation to a class without changing the class.

The Classic 23 Patterns introduced by **GoF**

Design patterns are beneficial for various reasons. They are proven solutions that industry veterans have tried and tested. They are solid approaches that solve issues in a widely accepted way and reflect the experience and insights of the industry-leading developers that helped define them. Patterns also make your code more reusable and readable while speeding up the development process vastly.

Design patterns are by no means finished solutions. They only provide us with approaches or schemes to solve a problem.

> ***Note:*** *In this article, we will mainly talk about design patterns from an object-oriented point of view and in the context of their usability in modern JavaScript. That is why many classic patterns from GoF may be omitted, and some modern patterns from sources like* ***Addy Osmani's Learn JavaScript Design Patterns*** *will be included. The examples are kept simple for easier understanding and are hence not the most optimised implementation of their respective design patterns.*

Categories of Design Patterns

Design patterns are usually categorized into three major groups.

Creational Design Patterns

As the name suggests, these patterns are for handling object creational mechanisms. A creational design pattern basically solves a problem by controlling the creation process of an object.

We will discuss the following patterns in detail: ***Constructor Pattern, Factory Pattern, Prototype Pattern,*** and ***Singleton Pattern.***

Structural Design Patterns

These patterns are concerned with class and object composition. They help structure or restructure one or more parts without affecting the entire system. In other words, they help obtain new functionalities without tampering with the existing ones.

We will discuss the following patterns in detail: ***Adapter Pattern, Composite Pattern, Decorator Pattern, Façade Pattern, Flyweight Pattern,*** and ***Proxy Pattern.***

Behavioral Design Patterns

These patterns are concerned with improving communication between dissimilar objects.

We will discuss the following patterns in detail: ***Chain of Responsibility Pattern, Command Pattern, Iterator Pattern, Mediator Pattern, Observer Pattern, State Pattern, Strategy Pattern,*** and ***Template Pattern.***

Constructor Pattern

This is a class-based creational design pattern. Constructors are special functions that can be used to instantiate new objects with methods and properties defined by that function.

It is not one of the classic design patterns. In fact, it is more of a basic language construct than a pattern in most object-oriented languages. But in JavaScript, objects can be created on the fly without any constructor functions or “class” definition. Therefore, I think it is important to lay down the foundation for other patterns to come with this simple one.

Constructor pattern is one of the most commonly used patterns in JavaScript for creating new objects of a given kind.

In this example, we define a `Hero` class with attributes like `name` and `specialAbility` and methods like `getDetails`. Then, we instantiate an object `IronMan` by invoking the constructor method with the `new` keyword passing in the values for the respective attributes as arguments.

```
// traditional Function-based syntax
function Hero(name, specialAbility) {
  // setting property values
  this.name = name;
  this.specialAbility = specialAbility;

  // declaring a method on the object
  this.getDetails = function() {
    return this.name + ' can ' + this.specialAbility;
  };
}

// ES6 Class syntax
class Hero {
  constructor(name, specialAbility) {
    // setting property values
    this._name = name;
    this._specialAbility = specialAbility;

    // declaring a method on the object
    this.getDetails = function() {
```

```
    this.getDetails = function() {
      return `${this._name} can ${this._specialAbility}`;
    };
  }
}

// creating new instances of Hero
const IronMan = new Hero('Iron Man', 'fly');

console.log(IronMan.getDetails()); // Iron Man can fly
```

Constructor.js hosted with ❤ by **GitHub** **view raw**

Constructor Pattern

Factory Pattern

Factory pattern is another class-based creational pattern. In this, we provide a generic interface that delegates the responsibility of object instantiation to its subclasses.

This pattern is frequently used when we need to manage or manipulate collections of objects that are different yet have many similar characteristics.

In this example, we create a factory class named `BallFactory` that has a method that takes in parameters, and, depending on the parameters, it delegates the object instantiation responsibility to the respective class. If the type parameter is `"football"` or `"soccer"` object instantiation is handled by `Football` class, but if it is `"basketball"` object instantiation is handled by `Basketball` class.

```
class BallFactory {
  constructor() {
    this.createBall = function(type) {
      let ball;
      if (type === 'football' || type === 'soccer') ball = new Football();
      else if (type === 'basketball') ball = new Basketball();
      ball.roll = function() {
        return `The ${this._type} is rolling.`;
      };

      return ball;
    };
  }
}

class Football {
  constructor() {
    this._type = 'football';
    this._type = 'football';
    this.kick = function() {
      return 'You kicked the football.';
    };
  }
}

class Basketball {
  constructor() {
    this._type = 'basketball';
    this.bounce = function() {
      return 'You bounced the basketball.';
    };
  }
}

// creating objects
const factory = new BallFactory();

const myFootball = factory.createBall('football');
const myBasketball = factory.createBall('basketball');

console.log(myFootball.roll()); // The football is rolling.
console.log(myBasketball.roll()); // The basketball is rolling.
console.log(myFootball.kick()); // You kicked the football.
console.log(myBasketball.bounce()); // You bounced the basketball.
```

Factory is hosted with ❤ by GitHub view raw

Factory Pattern

Prototype Pattern

This pattern is an object-based creational design pattern. In this, we use a sort of a “skeleton” of an existing object to create or instantiate new objects.

This pattern is specifically important and beneficial to JavaScript because it utilizes prototypal inheritance instead of a classic object-oriented inheritance. Hence, it plays to JavaScript’s strength and has native support.

In this example, we have a `car` object that we use as the prototype to create another object `myCar` with JavaScript’s `Object.create` feature and define an extra property `owner` on the new object.

```
// using Object.create as was recommended by ES5 standard
const car = {
  noOfWheels: 4,
  start() {
    return 'started';
  },
  stop() {
    return 'stopped';
  },
};

// Object.create(proto[, propertiesObject])

const myCar = Object.create(car, { owner: { value: 'John' } });

console.log(myCar.__proto__ === car); // true
```

Prototype.js hosted with ❤ by **GitHub** **view raw**

Prototype Pattern

Singleton Pattern

Singleton is a special creational design pattern in which only one instance of a class can exist. It works like this — if no instance of the singleton class exists then a new instance is created and returned, but if an instance already exists, then the reference to the existing instance is returned.

A perfect real-life example would be that of `mongoose` (the famous Node.js ODM library for MongoDB). It utilizes the singleton pattern.

In this example, we have a `Database` class that is a singleton. First, we create an object `mongo` by using the `new` operator to invoke the `Database` class constructor. This time an object is instantiated because none already exists. The second time, when we create the `mysql` object, no new object is instantiated but instead, the reference to the object that was instantiated earlier, i.e. the `mongo` object, is returned.

```
class Database {
  constructor(data) {
    if (Database.exists) {
      return Database.instance;
    }
    this._data = data;
    Database.instance = this;
    Database.exists = true;
    return this;
  }

  getData() {
    return this._data;
  }

  setData(data) {
    this._data = data;
  }
}

// usage
const mongo = new Database('mongo');
console.log(mongo.getData()); // mongo

const mysql = new Database('mysql');
console.log(mysql.getData()); // mongo
```

Singleton.js hosted with ❤ by **GitHub** **view raw**

Singleton Pattern

Adapter Pattern

This is a structural pattern where the interface of one class is translated into another. This pattern lets classes work together that could not otherwise because of incompatible interfaces.

This pattern is often used to create wrappers for new refactored APIs so that other existing old APIs can still work with them. This is usually done when new implementations or code refactoring (done for reasons like performance gains) result in a different public API, while the other parts of the system are still using the old API and need to be adapted to work together.

In this example, we have an old API, i.e. `OldCalculator` class, and a new API, i.e. `NewCalculator` class. The `OldCalculator` class provides an `operation` method for both addition and subtraction, while the `NewCalculator` provides separate methods for addition and subtraction. The Adapter class `CalcAdapter` wraps the `NewCalculator` to add the `operation` method to the public-facing API while using its own addition and subtraction implementation under the hood.

```
// old interface
class OldCalculator {
  constructor() {
    this.operations = function(term1, term2, operation) {
      switch (operation) {
        case 'add':
          return term1 + term2;
        case 'sub':
          return term1 - term2;
        default:
          return NaN;
      }
    };
  }
}

// new interface
class NewCalculator {
  constructor() {
    this.add = function(term1, term2) {
      return term1 + term2;
    };
    this.sub = function(term1, term2) {
      return term1 - term2;
    };
  }
}

// Adapter Class
class CalcAdapter {
  constructor() {
    const newCalc = new NewCalculator();

    this.operations = function(term1, term2, operation) {
      switch (operation) {
        case 'add':
          // using the new implementation under the hood
          return newCalc.add(term1, term2);
        case 'sub':
          return newCalc.sub(term1, term2);
        default:
          return NaN;
      }
    };
  }
}

// usage
const oldCalc = new OldCalculator();
console.log(oldCalc.operations(10, 5, 'add')); // 15

const newCalc = new NewCalculator();
console.log(newCalc.add(10, 5)); // 15
```

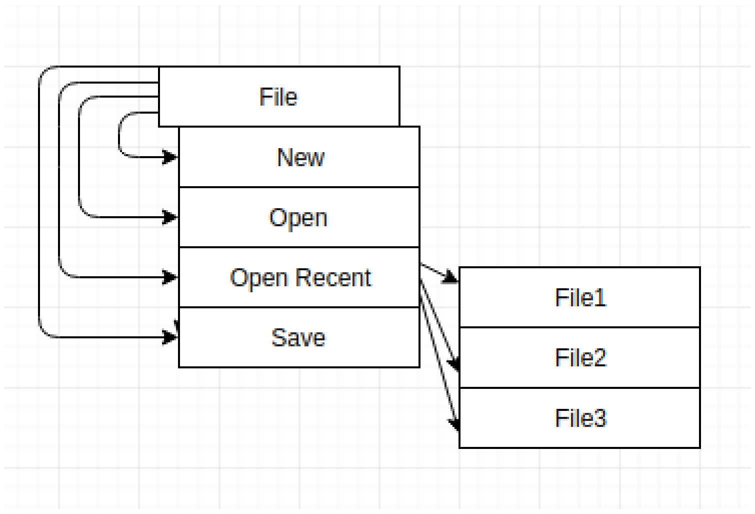
```javascript

const adaptedCalc = new CalcAdapter();
console.log(adaptedCalc.operations(10, 5, 'add')); // 15;
```

Adapter Pattern

Composite Pattern

This is a structural design pattern that composes objects into tree-like structures to represent whole-part hierarchies. In this pattern, each node in the tree-like structure can be either an individual object or a composed collection of objects. Regardless, each node is treated uniformly.



A Multi-level Menu Structure

It is a bit complex to visualize this pattern. The easiest way to think about this is with the example of a multi-level menu. Each node can be a distinct option, or it can be a menu itself, which has multiple options as its child. A node component with children is a composite component, while a node component without any child is a leaf component.

In this example, we create a base class of `Component` that implements the common functionalities needed and abstracts the other methods needed. The base class also has a static method that utilises recursion to traverse a composite tree structure made with its subclasses. Then we create two subclasses extending the base class — `Leaf` that does not have any children and `Composite` that can have children—and hence have methods handling adding, searching, and removing child functionalities. The two subclasses are used to create a composite structure—a tree, in this case.

```javascript
class Component {
  constructor(name) {
```

```
  constructor(name) {
    this._name = name;
  }

  getNodeName() {
    return this._name;
  }

  // abstract methods that need to be overridden
  getType() {}

  addChild(component) {}

  removeChildByName(componentName) {}

  removeChildByIndex(index) {}

  getChildByName(componentName) {}

  getChildByIndex(index) {}

  noOfChildren() {}

  static logTreeStructure(root) {
    let treeStructure = '';
    function traverse(node, indent = 0) {
      treeStructure += `${'--'.repeat(indent)}${node.getNodeName()}\n`;
      indent++;
      for (let i = 0, length = node.noOfChildren(); i < length; i++) {
        traverse(node.getChildByIndex(i), indent);
      }
    }

    traverse(root);
    return treeStructure;
  }
}

class Leaf extends Component {
  constructor(name) {
    super(name);
    this._type = 'Leaf Node';
  }

  getType() {
    return this._type;
  }

  noOfChildren() {
    return 0;
  }
}

class Composite extends Component {
  constructor(name) {
    super(name);
    this._type = 'Composite Node';
    this._children = [];
  }

  getType() {
    return this._type;
  }

  addChild(component) {
    this._children = [...this._children, component];
  }

  removeChildByName(componentName) {
    this._children = [...this._children].filter(component => component.getNodeName() !
  }

  removeChildByIndex(index) {
    this._children = [...this._children.slice(0, index), ...this._children.slice(index
  }

  getChildByName(componentName) {
    return this._children.find(component => component.name === componentName);
  }

  getChildByIndex(index) {
    return this._children[index];
  }

  noOfChildren() {
    return this._children.length;
  }
}

// usage
const tree = new Composite('root');
tree.addChild(new Leaf('left'));
const right = new Composite('right');
tree.addChild(right);
right.addChild(new Leaf('right-left'));
const rightMid = new Composite('right-middle');
const rightMid = new Composite('right-middle');
right.addChild(rightMid);
right.addChild(new Leaf('right-right'));
rightMid.addChild(new Leaf('left-end'));
rightMid.addChild(new Leaf('right-end'));

// log
console.log(Component.logTreeStructure(tree));
/*
root
--left
--right
----right-left
----right-middle
------left-end
------right-end
----right-right
*/
```

Decorator Pattern

This is also a structural design pattern that focuses on the ability to add behaviour or functionalities to existing classes dynamically. It is another viable alternative to sub-classing.

The decorator type behaviour is very easy to implement in JavaScript because JavaScript allows us to add methods and properties to object dynamically. The simplest approach would be to just add a property to an object, but it will not be efficiently reusable.

In fact, there is a proposal to add decorators to the JavaScript language. Take a look at **Addy Osmani's post** about decorators in JavaScript.

If you want to read about the **proposal itself**, feel free.

In this example, we create a `Book` class. We further create two decorator functions that accept a book object and return a "decorated" `book` object — `giftWrap` that adds one new attribute and one new function and `hardbindBook` that adds one new attribute and edits the value of one existing attribute.

```
class Book {
  constructor(title, author, price) {
    this._title = title;
    this._author = author;
    this.price = price;
  }

  getDetails() {
    return `${this._title} by ${this._author}`;
  }
}

// decorator 1
function giftWrap(book) {
  book.isGiftWrapped = true;
  book.unwrap = function() {
    return `Unwrapped ${book.getDetails()}`;
  };

  return book;
}

// decorator 2
function hardbindBook(book) {
  book.isHardbound = true;
  book.price += 5;
  return book;
}

// usage
const alchemist = giftWrap(new Book('The Alchemist', 'Paulo Coelho', 10));

console.log(alchemist.isGiftWrapped); // true
console.log(alchemist.unwrap()); // 'Unwrapped The Alchemist by Paulo Coelho'

const inferno = hardbindBook(new Book('Inferno', 'Dan Brown', 15));

console.log(inferno.isHardbound); // true
console.log(inferno.price); // 20
```

Decorator.js hosted with ❤ by **GitHub** **view raw**

Decorator Pattern

Façade Pattern

This is a structural design pattern that is widely used in the JavaScript libraries. It is used to provide a unified and simpler, public-facing interface for ease of use that shields away from the complexities of its consisting subsystems or subclasses.

The use of this pattern is very common in libraries like jQuery.

In this example, we create a public facing API with the class `ComplaintRegistry` . It exposes only one method to be used by the client, i.e. `registerComplaint` . It internally handles instantiating required objects of either `ProductComplaint` or `ServiceComplaint` based on the type argument. It also handles all the other complex functionalities like generating a unique ID, storing the complaint in memory, etc. But, all these complexities are hidden away using the façade pattern.

```
let currentId = 0;

class ComplaintRegistry {
  registerComplaint(customer, type, details) {
    const id = ComplaintRegistry._uniqueIdGenerator();
    let registry;
    if (type === 'service') {
      registry = new ServiceComplaints();
    } else {
      registry = new ProductComplaints();
    }
    return registry.addComplaint({ id, customer, details });
  }

  static _uniqueIdGenerator() {
    return ++currentId;
  }
}

class Complaints {
  constructor() {
    this.complaints = [];
  }

  addComplaint(complaint) {
    this.complaints.push(complaint);
    return this.replyMessage(complaint);
  }

  getComplaint(id) {
    return this.complaints.find(complaint => complaint.id === id);
  }

  replyMessage(complaint) {}
}

```

```
class ProductComplaints extends Complaints {
  constructor() {
    super();
    if (ProductComplaints.exists) {
      return ProductComplaints.instance;
    }
    ProductComplaints.instance = this;
    ProductComplaints.exists = true;
    return this;
  }

  replyMessage({ id, customer, details }) {
    return `Complaint No. ${id} reported by ${customer} regarding ${details} have been
  }
}

class ServiceComplaints extends Complaints {
  constructor() {
    super();
    if (ServiceComplaints.exists) {
      return ServiceComplaints.instance;
    }
    ServiceComplaints.instance = this;
    ServiceComplaints.exists = true;
    return this;
  }

  replyMessage({ id, customer, details }) {
    return `Complaint No. ${id} reported by ${customer} regarding ${details} have been
  }
}

// usage
const registry = new ComplaintRegistry();

const reportService = registry.registerComplaint('Martha', 'service', 'availability');
// 'Complaint No. 1 reported by Martha regarding availability have been filed with the

const reportProduct = registry.registerComplaint('Jane', 'product', 'faded color');
// 'Complaint No. 2 reported by Jane regarding faded color have been filed with the Pro
```

Flyweight Pattern

This is a structural design pattern focused on efficient data sharing through fine-grained objects. It is used for efficiency and memory conservation purposes.

This pattern can be used for any kind of caching purposes. In fact, modern browsers use a variant of a flyweight pattern to prevent loading the same images twice.

In this example, we create a fine-grained flyweight class `Icecream` for sharing data regarding ice-cream flavours and a factory class `IcecreamFactory` to create those flyweight objects. For memory conservation, the objects are recycled if the same object is instantiated twice. This is a simple example of flyweight implementation.

```
// flyweight class
class Icecream {
  constructor(flavour, price) {
    this.flavour = flavour;
    this.price = price;
  }
}

// factory for flyweight objects
class IcecreamFactory {
  constructor() {
    this._icecreams = [];
  }

  createIcecream(flavour, price) {
    let icecream = this.getIcecream(flavour);
    if (icecream) {
      return icecream;
    } else {
      const newIcecream = new Icecream(flavour, price);
      this._icecreams.push(newIcecream);
      return newIcecream;
    }
  }

  getIcecream(flavour) {
    return this._icecreams.find(icecream => icecream.flavour === flavour);
  }
}

// usage
const factory = new IcecreamFactory();

const chocoVanilla = factory.createIcecream('chocolate and vanilla', 15);
const vanillaChoco = factory.createIcecream('chocolate and vanilla', 15);

// reference to the same object
console.log(chocoVanilla === vanillaChoco); // true
```

Flyweight.js hosted with ❤ by GitHub view raw

Flyweight Pattern

Proxy Pattern

This is a structural design pattern that behaves exactly as its name suggests. It acts as a surrogate or placeholder for another object to control access to it.

It is usually used in situations in which a target object is under constraints and may not be able to handle all its responsibilities efficiently. A proxy, in this case, usually provides the same interface to the client and adds a level of indirection to support controlled access to the target object to avoid undue pressure on it.

The proxy pattern can be very useful when working with network request-heavy applications to avoid unnecessary or redundant network requests.

In this example, we will use two new ES6 features, **Proxy** and **Reflect**. A Proxy object is used to define custom behaviour for fundamental operations of a JavaScript object (remember, function and arrays are also object in JavaScript). It is a constructor method that can be used to create a `Proxy` object. It accepts a `target` object that is to be proxied and a `handler` object that will define the necessary customisation. The handler object allows for defining some trap functions like `get` , `set` , `has` , `apply` , etc. that are used to add custom behaviour attached to their usage. `Reflect` , on the other hand, is a built-in object that provides similar methods that are supported by the handler object of Proxy as static methods on itself. It is not a constructor; its static methods are used for intercept-able JavaScript operations.

Now, we create a function that can be thought of as a network request. We named it as `networkFetch` . It accepts a URL and responds accordingly. We want to implement a proxy where we only get the response from the network if it is not available in our cache. Otherwise, we just return a response from the cache.

The `cache` global variable will store our cached responses. We create a proxy named `proxiedNetworkFetch` with our original `networkFetch` as the `target` and use apply method in our `handler` object to proxy the function invocation. The apply method gets passed on the `target` object itself. This value as `thisArg` and the arguments are passed to it in an array-like structure `args` .

We check if the passed url argument is in the cache. If it exists in the cache, we return the response from there, never invoking the original target function. If it does not, then we use the `Reflect.apply` method to invoke the `target` function with `thisArg` (although it's not of any significance in our case here) and the arguments it passed.

```javascript
// Target
function networkFetch(url) {
  return `${url} - Response from network`;
}

// Proxy
// ES6 Proxy API = new Proxy(target, handler);
const cache = [];
const proxiedNetworkFetch = new Proxy(networkFetch, {
  apply(target, thisArg, args) {
    const urlParam = args[0];
    if (cache.includes(urlParam)) {
      return `${urlParam} - Response from cache`;
    } else {
      cache.push(urlParam);
      return Reflect.apply(target, thisArg, args);
    }
  },
});

// usage
console.log(proxiedNetworkFetch('dogPic.jpg')); // 'dogPic.jpg - Response from network
console.log(proxiedNetworkFetch('dogPic.jpg')); // 'dogPic.jpg - Response from cache'
```

Proxy.js hosted with ❤ by **GitHub** **view raw**

Proxy Pattern

Chain of Responsibility Pattern

This is a behavioural design pattern that provides a chain of loosely coupled objects. Each of these objects can choose to act on or handle the request of the client.

A good example of the chain of responsibility pattern is the event bubbling in DOM in which an event propagates through a series of nested DOM elements, one of which may have an "event listener" attached to listen to and act on the event.

In this example, we create a class `CumulativeSum`, which can be instantiated with an optional `initialValue`. It has a method `add` that adds the passed value to the `sum` attribute of the object and returns the `object` itself to allow chaining of `add` method calls.

This is a common pattern that can be seen in **jQuery** as well, where almost any method call on a jQuery object returns a jQuery object so that method calls can be chained together.

```
class CumulativeSum {
  constructor(intialValue = 0) {
    this.sum = intialValue;
  }

  add(value) {
    this.sum += value;
    return this;
  }
}

// usage
const sum1 = new CumulativeSum();
console.log(sum1.add(10).add(2).add(50).sum); // 62


const sum2 = new CumulativeSum(10);
console.log(sum2.add(10).add(20).add(5).sum); // 45
```

ChainOfResponsibility.js hosted with ❤ by GitHub view raw

Chain of Responsibility Pattern

Command Pattern

This is a behavioural design pattern that aims to encapsulate actions or operations as objects. This pattern allows loose coupling of systems and classes by separating the objects that request an operation or invoke a method from the ones that execute or process the actual implementation.

The clipboard interaction API somewhat resembles the command pattern. If you are a **Redux** user, you have already come across the command pattern. The actions that allow the awesome time-travel debugging feature are nothing but encapsulated operations that can be tracked to redo or undo operations. Hence, time-travelling made possible.

In this example, we have a class called `SpecialMath` that has multiple methods and a `Command` class that encapsulates commands that are to be executed on its subject, i.e. an object of the `SpecialMath` class. The `Command` class also keeps track of all the commands executed, which can be used to extend its functionality to include undo and redo type operations.

Command Pattern

Iterator Pattern

It is a behavioural design pattern that provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Iterators have a special kind of behaviour where we step through an ordered set of values one at a time by calling `next()` until we reach the end. The introduction of Iterator and Generators in ES6 made the implementation of the iterator pattern extremely straightforward.

We have two examples below. First, one `IteratorClass` uses iterator spec, while the other one `iteratorUsingGenerator` uses generator functions.

The `Symbol.iterator` (`Symbol` —a new kind of primitive data type) is used to specify the default iterator for an object. It must be defined for a collection to be able to use the `for...of` looping construct. In the first example, we define the constructor to store some collection of data and then define `Symbol.iterator` , which returns an object with `next` method for iteration.

For the second case, we define a generator function passing it an array of data and returning its elements iteratively using `next` and `yield` . A generator function is a special type of function that works as a factory for iterators and can explicitly maintain its own internal state and yield values iteratively. It can pause and resume its own execution cycle.

Iterator Pattern

Mediator Pattern

It is a behavioural design pattern that encapsulates how a set of objects interact with each other. It provides the central authority over a group of objects by promoting loose coupling, keeping objects from referring to each other explicitly.

In this example, we have `TrafficTower` as Mediator that controls the way `Airplane` objects interact with each other. All the `Airplane` objects register themselves with a `TrafficTower` object, and it is the mediator class object that handles how an `Airplane` object receives coordinates data of all the other `Airplane` objects.

Mediator Pattern

Observer Pattern

It is a crucial behavioural design pattern that defines one-to-many dependencies between objects so that when one object (publisher) changes its state, all the other dependent objects (subscribers) are notified and updated automatically. This is also called PubSub (publisher/subscribers) or event dispatcher/listeners pattern. The publisher is sometimes called the subject, and the subscribers are sometimes called observers.

Chances are, you're already somewhat familiar with this pattern if you have used `addEventListener` or jQuery's `.on` to write even-handling code. It has its influences in Reactive Programming (think **RxJS**) as well.

In the example, we create a simple `Subject` class that has methods to add and remove objects of `Observer` class from subscriber collection. Also, a `fire` method to propagate any changes in the `Subject` class object to the subscribed Observers. The `Observer` class, on the other hand, has its internal state and a method to update its internal state based on the change propagated from the `Subject` it has subscribed to.

Observer Pattern

State Pattern

It is a behavioural design pattern that allows an object to alter its behaviour based on changes to its internal state. The object returned by a state pattern class seems to change its class. It provides state-specific logic to a limited set of objects in which each object type represents a particular state.

We will take a simple example of a traffic light to understand this pattern. The `TrafficLight` class changes the object it returns based on its internal state, which is an object of `Red` , `Yellow` , or `Green` class.

State Pattern

Strategy Pattern

It is a behavioural design pattern that allows encapsulation of alternative algorithms for a particular task. It defines a family of algorithms and encapsulates them in such a way that they are interchangeable at runtime without client interference or knowledge.

In the example below, we create a class `Commute` for encapsulating all the possible strategies for commuting to work. Then, we define three strategies namely `Bus` , `PersonalCar` , and `Taxi` . Using this pattern we can swap the implementation to use for the `travel` method of the `Commute` object at runtime.

Strategy Pattern

Template Pattern

This is a behavioural design pattern based on defining the skeleton of the algorithm or implementation of an operation, but deferring some steps to subclasses. It lets subclasses redefine certain steps of an algorithm without changing the algorithm's outward structure.

In this example, we have a Template class `Employee` that implements `work` method partially. It is for the subclasses to implement responsibilities method to make it work as a whole. We then create two subclasses `Developer` and `Tester` that extend the template class and implement the required method to fill the implementation gap.

Template Pattern

Conclusion

Design patterns are crucial to software engineering and can be very helpful in solving common problems. But this is a very vast subject, and it is simply not possible to include everything about them in a short piece. Therefore, I made the choice to shortly and concisely talk only about the ones I think can be really handy in writing modern JavaScript. To dive deeper, I suggest you take a look at these books:

1. **Design Patterns: Elements Of Reusable Object-Oriented Software** by *Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides (Gang of Four)*

2. **Learn JavaScript Design Patterns** by *Addy Osmani*

3. **JavaScript Patterns** by *Stoyan Stefanov*