JavaScript Design Patterns Guide

Table of Contents

Creational Patterns

Constructor Pattern

Concept: Creates new objects with their own object scope. In JavaScript, constructor functions are used to create specific types of objects with the `new` keyword.

Key JavaScript Concepts:

- Constructors
- Prototypal inheritance
- `this` binding
- The `new` keyword

Example 1: Basic Constructor

```
// Constructor function
function Person(name, age) {
  // Properties assigned to the object using 'this'
  this.name = name;
  this.age = age;

  // Method defined within the constructor
  this.greet = function() {
    return `Hello, my name is ${this.name} and I am ${this.age} years old`;
  };
}

// Creating new instances with the 'new' keyword
const person1 = new Person('Alice', 25);
const person2 = new Person('Bob', 30);

console.log(person1.greet()); // "Hello, my name is Alice and I am 25 years old"
console.log(person2.greet()); // "Hello, my name is Bob and I am 30 years old"
```

Example 2: Constructor with Prototype for Better Memory Efficiency

```
// Constructor function
function Vehicle(make, model, year) {
  // Instance properties
  this.make = make;
  this.model = model;
  this.year = year;
  this.isRunning = false;
}

// Methods added to the prototype (shared across all instances)
// This is more memory efficient than defining methods in the constructor
Vehicle.prototype.startEngine = function() {
  this.isRunning = true;
  return `${this.make} ${this.model}'s engine is started`;
};

Vehicle.prototype.stopEngine = function() {
  this.isRunning = false;
  return `${this.make} ${this.model}'s engine is stopped`;
};

Vehicle.prototype.honk = function() {
  return 'Beep beep!';
};

// Create instances
const car1 = new Vehicle('Toyota', 'Corolla', 2020);
const car2 = new Vehicle('Honda', 'Civic', 2021);

console.log(car1.startEngine()); // "Toyota Corolla's engine is started"
console.log(car2.honk());        // "Beep beep!"

// All instances share the same method definitions
console.log(car1.honk === car2.honk); // true - methods reference the same function
```

Module Pattern

Concept: Uses closures to create private and public encapsulation. This pattern emulates the concept of classes with private and public methods and variables.

Key JavaScript Concepts:

- Closures
- IIFEs (Immediately Invoked Function Expressions)
- Private and public scope

Example 1: Basic Module Pattern

```
// Module pattern using an IIFE to create encapsulation
const Counter = (function() {
  // Private variables - not accessible outside the module
  let count = 0;

  // Private function
  const validate = function(num) {
    return typeof num === 'number' && !isNaN(num);
  };

  // Return an object with public methods
  return {
    // Public methods that can access private variables
    increment: function() {
      return ++count;
    },
    decrement: function() {
      return --count;
    },
    getValue: function() {
      return count;
    },
    setValue: function(value) {
      if (validate(value)) {
        count = value;
        return true;
      }
      return false;
    }
  };
})();

console.log(Counter.getValue()); // 0
Counter.increment();
Counter.increment();
console.log(Counter.getValue()); // 2
Counter.setValue(10);
console.log(Counter.getValue()); // 10
// count and validate are not accessible directly
// console.log(Counter.count); // undefined
```

Example 2: Revealing Module Pattern

```
// Revealing Module Pattern - defines all functions privately and exposes only what's n
const Calculator = (function() {
  // Private variables and functions
  let result = 0;

  function add(x, y) {
    return x + y;
  }

  function subtract(x, y) {
    return x - y;
  }

  function multiply(x, y) {
    return x * y;
  }

  function divide(x, y) {
    if (y === 0) {
      throw new Error("Cannot divide by zero");
    }
    return x / y;
  }

  function setResult(value) {
    result = value;
    return result;
  }

  function getResult() {
    return result;
  }

  // Only reveal the public methods and properties
  return {
    add: add,
    subtract: subtract,
    multiply: multiply,
    divide: divide,
    setResult: setResult,
    result: getResult // Note this maps to the getResult function, not the variable dir
  };
})();
```

```
console.log(Calculator.add(5, 3));       // 8
Calculator.setResult(Calculator.add(10, 5));
console.log(Calculator.result());        // 15
console.log(Calculator.multiply(4, 2));  // 8
```

Factory Pattern

Concept: Creates objects without specifying the exact class or constructor function of the object. It provides a generic interface for creating objects.

Key JavaScript Concepts:

- Factory functions
- Object creation abstraction
- Dynamic object initialization

Example 1: Basic Factory Function

```
// Factory function for creating different types of vehicles
function createVehicle(type, attributes) {
  // Base vehicle object with shared properties and methods
  const vehicle = {
    type: type,
    start() {
      return `${this.type} started`;
    },
    stop() {
      return `${this.type} stopped`;
    }
  };

  // Extend vehicle based on type
  if (type === 'car') {
    vehicle.wheels = 4;
    vehicle.doors = attributes.doors || 4;
    vehicle.honk = function() {
      return 'Beep beep!';
    };
  }
  else if (type === 'motorcycle') {
    vehicle.wheels = 2;
    vehicle.hasHelmet = attributes.hasHelmet || false;
    vehicle.wheelie = function() {
      return 'Doing a wheelie!';
    };
  }
  else if (type === 'bicycle') {
    vehicle.wheels = 2;
    vehicle.hasBell = attributes.hasBell || true;
    vehicle.ringBell = function() {
      return 'Ring ring!';
    };
  }

  return vehicle;
}

// Create different vehicles
const car = createVehicle('car', { doors: 2 });
const motorcycle = createVehicle('motorcycle', { hasHelmet: true });
const bicycle = createVehicle('bicycle', { hasBell: false });
```

```
console.log(car.start());           // "car started"
console.log(car.honk());            // "Beep beep!"
console.log(motorcycle.wheelie());  // "Doing a wheelie!"
console.log(bicycle.ringBell());    // "Ring ring!"
```

Example 2: Factory Using Class Inheritance

```
// Base Vehicle class
class Vehicle {
  constructor(options) {
    this.make = options.make || 'Default';
    this.model = options.model || 'Default';
  }

  getInfo() {
    return `${this.make} ${this.model}`;
  }

  start() {
    return `${this.getInfo()} is starting`;
  }
}

// Car subclass
class Car extends Vehicle {
  constructor(options) {
    super(options);
    this.doors = options.doors || 4;
    this.isElectric = options.isElectric || false;
  }

  honk() {
    return `${this.getInfo()} goes beep!`;
  }
}

// Truck subclass
class Truck extends Vehicle {
  constructor(options) {
    super(options);
    this.bedSize = options.bedSize || 'standard';
    this.towingCapacity = options.towingCapacity || 5000;
  }

  loadCargo(amount) {
    return `${this.getInfo()} is loading ${amount} kg of cargo`;
  }
}

// Vehicle Factory
```

```
class VehicleFactory {
  static createVehicle(type, options) {
    switch(type) {
      case 'car':
        return new Car(options);
      case 'truck':
        return new Truck(options);
      default:
        return new Vehicle(options);
    }
  }
}

// Create vehicles using the factory
const sedan = VehicleFactory.createVehicle('car', {
  make: 'Honda',
  model: 'Accord',
  doors: 4
});

const pickup = VehicleFactory.createVehicle('truck', {
  make: 'Ford',
  model: 'F-150',
  bedSize: 'large',
  towingCapacity: 7500
});

console.log(sedan.start());         // "Honda Accord is starting"
console.log(sedan.honk());          // "Honda Accord goes beep!"
console.log(pickup.loadCargo(500)); // "Ford F-150 is loading 500 kg of cargo"
```

Singleton Pattern

Concept: Restricts instantiation of a class to a single instance and provides a global point of access to it. Useful when exactly one object is needed across the system.

Key JavaScript Concepts:

- Closures for private state
- Lazy initialization
- Global access point

Example 1: Classic Singleton with IIFE

```
// Singleton using an IIFE and closure
const Database = (function() {
  // Private variable to store the instance
  let instance;

  // Private constructor function
  function createInstance() {
    // Private members
    const _data = [];
    const _dbName = "SingletonDB";

    // Public interface
    return {
      // Add data to the database
      add: function(item) {
        _data.push(item);
        return _data.length;
      },
      // Get data by index
      get: function(index) {
        return _data[index];
      },
      // Get all data
      getAll: function() {
        return [..._data]; // Return a copy to prevent external modification
      },
      // Get database name
      getName: function() {
        return _dbName;
      }
    };
  }

  // Return the public interface
  return {
    // Method to get the instance
    getInstance: function() {
      // Create the instance if it doesn't exist
      if (!instance) {
        instance = createInstance();
      }
      return instance;
    }
```

```
  };
})();

// Usage
const db1 = Database.getInstance();
const db2 = Database.getInstance();

console.log(db1 === db2); // true - both variables reference the same instance

db1.add("First item");
db1.add("Second item");

console.log(db2.getAll()); // ["First item", "Second item"]
console.log(db1.getName()); // "SingletonDB"
```

Example 2: ES6 Singleton with Class

```
// Modern ES6 implementation of Singleton
class ConfigManager {
  constructor() {
    // Ensure instance is created only once
    if (ConfigManager.instance) {
      return ConfigManager.instance;
    }

    // Initialize the singleton instance
    this.config = {
      apiUrl: 'https://api.example.com',
      timeout: 3000,
      version: '1.0.0'
    };

    // Store the instance
    ConfigManager.instance = this;
  }

  // Method to get a config value
  get(key) {
    return this.config[key];
  }

  // Method to set a config value
  set(key, value) {
    this.config[key] = value;
    return this.config[key];
  }

  // Get all config values
  getAll() {
    return { ...this.config }; // Return a copy to prevent external modification
  }
}

// Usage
const config1 = new ConfigManager();
const config2 = new ConfigManager();

console.log(config1 === config2); // true - both are the same instance

config1.set('timeout', 5000);
```

```
console.log(config2.get('timeout')); // 5000 - reflects change made via config1

// We can also create a static getInstance method for more traditional singleton access
ConfigManager.getInstance = function() {
  return new ConfigManager(); // Constructor ensures only one instance
};

const config3 = ConfigManager.getInstance();
console.log(config1 === config3); // true
```

Prototype Pattern

Concept: Creates objects based on a template of an existing object through cloning. This pattern is used when object creation is costly.

Key JavaScript Concepts:

- Prototypal inheritance
- Object cloning
- Object.create()

Example 1: Basic Prototype Pattern

```
// Base prototype object with common properties and methods
const carPrototype = {
  init(make, model, year) {
    this.make = make;
    this.model = model;
    this.year = year;
    return this;
  },
  getInfo() {
    return `${this.year} ${this.make} ${this.model}`;
  },
  start() {
    return `${this.getInfo()} is starting`;
  },
  stop() {
    return `${this.getInfo()} is stopping`;
  }
};

// Car factory that uses the prototype
const carFactory = {
  // Create a new car based on the prototype
  createCar(make, model, year) {
    // Object.create() creates a new object with the specified prototype
    return Object.create(carPrototype).init(make, model, year);
  }
};

// Create cars using the factory
const car1 = carFactory.createCar('Toyota', 'Camry', 2020);
const car2 = carFactory.createCar('Honda', 'Civic', 2021);

console.log(car1.getInfo()); // "2020 Toyota Camry"
console.log(car2.start());   // "2021 Honda Civic is starting"

// Both objects share the same methods through prototype linking
console.log(car1.start === car2.start); // true
```

Example 2: Prototype Pattern with Clone Method

```
// Employee prototype object
const employeePrototype = {
  init(name, role, salary) {
    this.name = name;
    this.role = role;
    this.salary = salary;
    return this;
  },

  // Method to create a clone of the employee
  clone() {
    // Create a new object that inherits from the same prototype
    const clone = Object.create(Object.getPrototypeOf(this));

    // Copy own properties
    const props = Object.getOwnPropertyNames(this);
    props.forEach(prop => {
      const desc = Object.getOwnPropertyDescriptor(this, prop);
      Object.defineProperty(clone, prop, desc);
    });

    return clone;
  },

  // Method to give a raise
  giveRaise(amount) {
    this.salary += amount;
    return this.salary;
  },

  // Method to promote to a new role
  promote(newRole) {
    this.role = newRole;
    return this.role;
  },

  // Method to get employee info
  getInfo() {
    return `${this.name} - ${this.role}, $${this.salary}`;
  }
};

// Create a base employee
```

```
const baseEmployee = Object.create(employeePrototype).init('', '', 0);

// Create a developer based on the base employee
const developer = baseEmployee.clone();
developer.init('John Doe', 'Junior Developer', 70000);

// Create a designer based on the developer but change properties
const designer = developer.clone();
designer.name = 'Jane Smith';
designer.role = 'UI Designer';
designer.salary = 75000;

// Create a senior dev by cloning and modifying
const seniorDev = developer.clone();
seniorDev.name = 'Bob Johnson';
seniorDev.promote('Senior Developer');
seniorDev.giveRaise(30000);

console.log(developer.getInfo()); // "John Doe - Junior Developer, $70000"
console.log(designer.getInfo());  // "Jane Smith - UI Designer, $75000"
console.log(seniorDev.getInfo()); // "Bob Johnson - Senior Developer, $100000"

// All objects share the same methods
console.log(developer.getInfo === designer.getInfo); // true
```

Structural Patterns

Adapter Pattern

Concept: Allows objects with incompatible interfaces to work together by creating a wrapper (adapter) that translates one interface to another.

Key JavaScript Concepts:

- Interface conversion
- Legacy system integration
- Wrapper functions

Example 1: Basic Adapter

```
// Old API - uses meters
const MetricCalculator = {
  calculateArea(length, width) {
    return length * width;
  },
  calculateVolume(length, width, height) {
    return length * width * height;
  }
};

// We need an adapter to work with imperial units (feet)
const ImperialAdapter = {
  // Conversion rate from feet to meters
  FEET_TO_METERS: 0.3048,

  // Adapt the calculateArea method to work with feet
  calculateArea(lengthInFeet, widthInFeet) {
    // Convert feet to meters
    const lengthInMeters = lengthInFeet * this.FEET_TO_METERS;
    const widthInMeters = widthInFeet * this.FEET_TO_METERS;

    // Use the old API with converted values
    return MetricCalculator.calculateArea(lengthInMeters, widthInMeters);
  },

  // Adapt the calculateVolume method to work with feet
  calculateVolume(lengthInFeet, widthInFeet, heightInFeet) {
    // Convert feet to meters
    const lengthInMeters = lengthInFeet * this.FEET_TO_METERS;
    const widthInMeters = widthInFeet * this.FEET_TO_METERS;
    const heightInMeters = heightInFeet * this.FEET_TO_METERS;

    // Use the old API with converted values
    return MetricCalculator.calculateVolume(lengthInMeters, widthInMeters, heightInMete
  }
};

// Client code using the adapter
const roomSizeInFeet = {
  length: 15,
  width: 10
};
```

```
// Calculate area using the adapter
const areaInSquareMeters = ImperialAdapter.calculateArea(
  roomSizeInFeet.length,
  roomSizeInFeet.width
);

console.log(`Area: ${areaInSquareMeters.toFixed(2)} square meters`);
// Output: "Area: 4.47 square meters"
```

Example 2: Class-Based Adapter Pattern

```
// Modern API with async/await
class ModernAuthService {
  async authenticate(username, password) {
    // Simulate API call
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        if (username === 'admin' && password === 'password') {
          resolve({
            success: true,
            token: 'jwt-token-12345',
            user: { id: 1, username, role: 'admin' }
          });
        } else {
          reject(new Error('Authentication failed'));
        }
      }, 100);
    });
  }

  async validateToken(token) {
    // Simulate token validation
    return new Promise((resolve) => {
      setTimeout(() => {
        resolve(token === 'jwt-token-12345');
      }, 50);
    });
  }
}

// Legacy API with callbacks
class LegacyAuthService {
  login(username, password, callback) {
    // Simulate legacy API call
    setTimeout(() => {
      if (username === 'admin' && password === 'password') {
        callback(null, {
          status: 'success',
          apiKey: 'api-key-67890',
          userData: { userId: 1, name: username, permissions: ['admin'] }
        });
      } else {
        callback(new Error('Login failed'));
      }
```

```
    }, 100);
  }

  checkApiKey(apiKey, callback) {
    // Simulate legacy API key check
    setTimeout(() => {
      const isValid = apiKey === 'api-key-67890';
      callback(null, isValid);
    }, 50);
  }
}

// Adapter to make LegacyAuthService work with Modern interface
class AuthServiceAdapter {
  constructor(legacyService) {
    this.legacyService = legacyService;
  }

  // Adapt legacy login to return a Promise
  async authenticate(username, password) {
    return new Promise((resolve, reject) => {
      this.legacyService.login(username, password, (error, result) => {
        if (error) {
          reject(error);
          return;
        }

        // Transform legacy response format to match modern API
        resolve({
          success: result.status === 'success',
          token: result.apiKey, // Use apiKey as token
          user: {
            id: result.userData.userId,
            username: result.userData.name,
            role: result.userData.permissions[0]
          }
        });
      });
    });
  }

  // Adapt legacy checkApiKey to return a Promise
  async validateToken(token) {
```

```
    return new Promise((resolve, reject) => {
      this.legacyService.checkApiKey(token, (error, isValid) => {
        if (error) {
          reject(error);
          return;
        }
        resolve(isValid);
      });
    });
  }
}

// Usage example
async function testAuth() {
  // Modern service
  const modernAuth = new ModernAuthService();
  console.log('Testing modern auth service:');
  try {
    const modernResult = await modernAuth.authenticate('admin', 'password');
    console.log('Modern auth result:', modernResult);
    const isValidToken = await modernAuth.validateToken(modernResult.token);
    console.log('Token valid:', isValidToken);
  } catch (error) {
    console.error('Modern auth error:', error);
  }

  // Adapted legacy service
  const legacyAuth = new LegacyAuthService();
  const adaptedAuth = new AuthServiceAdapter(legacyAuth);

  console.log('\nTesting adapted legacy auth service:');
  try {
    // We can use the same interface as the modern service
    const adaptedResult = await adaptedAuth.authenticate('admin', 'password');
    console.log('Adapted auth result:', adaptedResult);
    const isValidToken = await adaptedAuth.validateToken(adaptedResult.token);
    console.log('Token valid:', isValidToken);
  } catch (error) {
    console.error('Adapted auth error:', error);
  }
}
```

```
// Run the test
testAuth();
```

Decorator Pattern

Concept: Dynamically adds behavior to an object without affecting the behavior of other objects from the same class. Uses composition instead of inheritance.

Key JavaScript Concepts:

- Function composition
- Object extension
- Dynamic behavior addition
- Higher-order functions

Example 1: Function Decorators

```
// Base function we want to decorate
function calculatePrice(price) {
  return price;
}

// Decorator 1: Add tax (15%)
function withTax(originalFunction) {
  // Return a new function that wraps the original
  return function(price) {
    // Call the original function
    const originalResult = originalFunction(price);
    // Add 15% tax
    const withTax = originalResult * 1.15;
    return parseFloat(withTax.toFixed(2));
  };
}

// Decorator 2: Add shipping ($5)
function withShipping(originalFunction) {
  return function(price) {
    const originalResult = originalFunction(price);
    // Add $5 shipping
    return parseFloat((originalResult + 5).toFixed(2));
  };
}

// Decorator 3: Apply discount (10%)
function withDiscount(originalFunction) {
  return function(price) {
    const originalResult = originalFunction(price);
    // Apply 10% discount
    const withDiscount = originalResult * 0.9;
    return parseFloat(withDiscount.toFixed(2));
  };
}

// Create decorated functions by composing decorators
// Order matters! Inner functions execute first
const calculatePriceWithTax = withTax(calculatePrice);
const calculatePriceWithShipping = withShipping(calculatePrice);
const calculatePriceWithDiscountAndTax = withTax(withDiscount(calculatePrice));
const calculateFinalPrice = withShipping(withTax(calculatePrice));
const calculateSpecialPrice = withDiscount(withShipping(withTax(calculatePrice)));
```

```
// Test the decorated functions
const basePrice = 100;
console.log(`Base Price: $${calculatePrice(basePrice)}`);                 // $100
console.log(`With Tax: $${calculatePriceWithTax(basePrice)}`);            // $115
console.log(`With Shipping: $${calculatePriceWithShipping(basePrice)}`);  // $105
console.log(`With Discount and Tax: $${calculatePriceWithDiscountAndTax(basePrice)}`);
console.log(`Final Price (Tax + Shipping): $${calculateFinalPrice(basePrice)}`);
console.log(`Special Price (Tax + Shipping + Discount): $${calculateSpecialPrice(basePr
```

Example 2: Object Decorators

```
// Base coffee object
class Coffee {
  constructor() {
    this.description = 'Basic Coffee';
    this.cost = 2;
  }

  getDescription() {
    return this.description;
  }

  getCost() {
    return this.cost;
  }
}

// Decorator base class
class CoffeeDecorator {
  constructor(coffee) {
    this.coffee = coffee;
  }

  getDescription() {
    return this.coffee.getDescription();
  }

  getCost() {
    return this.coffee.getCost();
  }
}

// Concrete decorators
class MilkDecorator extends CoffeeDecorator {
  constructor(coffee) {
    super(coffee);
    this.description = 'Milk';
    this.cost = 0.5;
  }

  getDescription() {
    return `${this.coffee.getDescription()}, ${this.description}`;
  }
```

```
  getCost() {
    return this.coffee.getCost() + this.cost;
  }
}

class WhippedCreamDecorator extends CoffeeDecorator {
  constructor(coffee) {
    super(coffee);
    this.description = 'Whipped Cream';
    this.cost = 0.7;
  }

  getDescription() {
    return `${this.coffee.getDescription()}, ${this.description}`;
  }

  getCost() {
    return this.coffee.getCost() + this.cost;
  }
}

class ChocolateDecorator extends CoffeeDecorator {
  constructor(coffee) {
    super(coffee);
    this.description = 'Chocolate';
    this.cost = 0.6;
  }

  getDescription() {
    return `${this.coffee.getDescription()}, ${this.description}`;
  }

  getCost() {
    return this.coffee.getCost() + this.cost;
  }
}

class ExtraShotDecorator extends CoffeeDecorator {
  constructor(coffee) {
    super(coffee);
    this.description = 'Extra Shot';
    this.cost = 1.1;
  }
```

```
  getDescription() {
    return `${this.coffee.getDescription()}, ${this.description}`;
  }

  getCost() {
    return this.coffee.getCost() + this.cost;
  }
}

// Create some coffee orders using decorators
let coffee = new Coffee();
console.log(`Order 1: ${coffee.getDescription()} - $${coffee.getCost()}`);
// "Order 1: Basic Coffee - $2"

// Decorate with milk
coffee = new MilkDecorator(coffee);
console.log(`Order 2: ${coffee.getDescription()} - $${coffee.getCost()}`);
// "Order 2: Basic Coffee, Milk - $2.5"

// Add whipped cream to the coffee with milk
coffee = new WhippedCreamDecorator(coffee);
console.log(`Order 3: ${coffee.getDescription()} - $${coffee.getCost()}`);
// "Order 3: Basic Coffee, Milk, Whipped Cream - $3.2"

// Create a more complex coffee from scratch
let specialCoffee = new Coffee();
specialCoffee = new ChocolateDecorator(specialCoffee);
specialCoffee = new WhippedCreamDecorator(specialCoffee);
specialCoffee = new ExtraShotDecorator(specialCoffee);
console.log(`Special: ${specialCoffee.getDescription()} - $${specialCoffee.getCost()}`)
// "Special: Basic Coffee, Chocolate, Whipped Cream, Extra Shot - $4.4"
```

Facade Pattern

Concept: Provides a simplified interface to a complex system. It hides the complexities of the underlying system and provides a cleaner API to the client.

Key JavaScript Concepts:

- API simplification
- Abstraction of complex systems

- Decoupling client code from subsystems

Example 1: Home Theater Facade

```
// Subsystem components
class DVDPlayer {
  constructor() {
    this.movie = null;
  }

  on() {
    return "DVD player is on";
  }

  off() {
    return "DVD player is off";
  }

  play(movie) {
    this.movie = movie;
    return `Playing "${movie}" on DVD player`;
  }

  stop() {
    const currentMovie = this.movie;
    this.movie = null;
    return `Stopped playing "${currentMovie}"`;
  }
}

class Amplifier {
  constructor() {
    this.volume = 0;
  }

  on() {
    return "Amplifier is on";
  }

  off() {
    return "Amplifier is off";
  }

  setVolume(level) {
    this.volume = level;
    return `Amplifier volume set to ${level}`;
  }
```

```
}

class Lights {
  on() {
    return "Lights are on";
  }

  off() {
    return "Lights are off";
  }

  dim(level) {
    return `Lights dimmed to ${level}%`;
  }
}

class Screen {
  down() {
    return "Screen is down";
  }

  up() {
    return "Screen is up";
  }
}

class Projector {
  on() {
    return "Projector is on";
  }

  off() {
    return "Projector is off";
  }

  setInput(input) {
    return `Projector input set to ${input}`;
  }
}

// Facade for the home theater system
class HomeTheaterFacade {
  constructor(dvdPlayer, amplifier, lights, screen, projector) {
```

```
    this.dvdPlayer = dvdPlayer;
    this.amplifier = amplifier;
    this.lights = lights;
    this.screen = screen;
    this.projector = projector;
  }

  // Simplified method to watch a movie
  watchMovie(movie) {
    console.log("Get ready to watch a movie...");
    // All the steps required to set up the theater
    const steps = [
      this.lights.dim(10),
      this.screen.down(),
      this.projector.on(),
      this.projector.setInput("DVD"),
      this.amplifier.on(),
      this.amplifier.setVolume(5),
      this.dvdPlayer.on(),
      this.dvdPlayer.play(movie)
    ];

    return steps.join("\n");
  }

  // Simplified method to end a movie
  endMovie() {
    console.log("Shutting down the theater...");
    // All the steps required to shut down the theater
    const steps = [
      this.dvdPlayer.stop(),
      this.dvdPlayer.off(),
      this.amplifier.off(),
      this.projector.off(),
      this.screen.up(),
      this.lights.on()
    ];

    return steps.join("\n");
  }
}

// Client code
```

```
const dvdPlayer = new DVDPlayer();
const amplifier = new Amplifier();
const lights = new Lights();
const screen = new Screen();
const projector = new Projector();

// Create the facade
const homeTheater = new HomeTheaterFacade(
  dvdPlayer, amplifier, lights, screen, projector
);

// Use the simplified interface
console.log(homeTheater.watchMovie("Inception"));
// Get ready to watch a movie...
// Lights dimmed to 10%
// Screen is down
// Projector is on
// Projector input set to DVD
// Amplifier is on
// Amplifier volume set to 5
// DVD player is on
// Playing "Inception" on DVD player

console.log(homeTheater.endMovie());
// Shutting down the theater...
// Stopped playing "Inception"
// DVD player is off
// Amplifier is off
// Projector is off
// Screen is up
// Lights are on
```

Example 2: API Facade

```
// Complex subsystems for API interactions
class RestApiService {
  constructor(baseUrl) {
    this.baseUrl = baseUrl;
    this.headers = { 'Content-Type': 'application/json' };
  }

  async get(endpoint, queryParams = {}) {
    const url = new URL(`${this.baseUrl}/${endpoint}`);
    Object.keys(queryParams).forEach(key => {
      url.searchParams.append(key, queryParams[key]);
    });

    try {
      const response = await fetch(url, { headers: this.headers });
      if (!response.ok) {
        throw new Error(`HTTP error ${response.status}`);
      }
      return await response.json();
    } catch (error) {
      console.error(`GET request failed: ${error.message}`);
      throw error;
    }
  }

  async post(endpoint, data) {
    try {
      const response = await fetch(`${this.baseUrl}/${endpoint}`, {
        method: 'POST',
        headers: this.headers,
        body: JSON.stringify(data)
      });

      if (!response.ok) {
        throw new Error(`HTTP error ${response.status}`);
      }
      return await response.json();
    } catch (error) {
      console.error(`POST request failed: ${error.message}`);
      throw error;
    }
  }
```

```
  // Other methods like put, delete, etc.
}

class AuthenticationService {
  constructor(apiService) {
    this.apiService = apiService;
    this.token = null;
  }

  async login(username, password) {
    try {
      const response = await this.apiService.post('auth/login', { username, password })
      this.token = response.token;
      localStorage.setItem('token', this.token);
      return response;
    } catch (error) {
      console.error('Login failed:', error);
      throw error;
    }
  }

  logout() {
    this.token = null;
    localStorage.removeItem('token');
    return { success: true };
  }

  isAuthenticated() {
    return !!this.token;
  }
}

class UserService {
  constructor(apiService) {
    this.apiService = apiService;
  }

  async getUserProfile(userId) {
    return await this.apiService.get(`users/${userId}`);
  }

  async updateUserProfile(userId, profileData) {
    return await this.apiService.post(`users/${userId}/update`, profileData);
```

```
  }
}

class ProductService {
  constructor(apiService) {
    this.apiService = apiService;
  }

  async getProducts(filters = {}) {
    return await this.apiService.get('products', filters);
  }

  async getProductDetails(productId) {
    return await this.apiService.get(`products/${productId}`);
  }
}

// Facade for the API interactions
class AppFacade {
  constructor(baseUrl = 'https://api.example.com') {
    // Create the core API service
    this.apiService = new RestApiService(baseUrl);

    // Create the subsystem services
    this.authService = new AuthenticationService(this.apiService);
    this.userService = new UserService(this.apiService);
    this.productService = new ProductService(this.apiService);
  }

  // Authentication methods
  async login(username, password) {
    return await this.authService.login(username, password);
  }

  logout() {
    return this.authService.logout();
  }

  isLoggedIn() {
    return this.authService.isAuthenticated();
  }

  // User methods
```

```
  async getUserProfile(userId) {
    return await this.userService.getUserProfile(userId);
  }

  async updateProfile(userId, profileData) {
    return await this.userService.updateUserProfile(userId, profileData);
  }

  // Product methods
  async getProducts(filters) {
    return await this.productService.getProducts(filters);
  }

  async getProductDetails(productId) {
    return await this.productService.getProductDetails(productId);
  }
}

// Client code using the facade
async function appDemo() {
  // Create the facade
  const app = new AppFacade();

  try {
    // Login
    await app.login('user123', 'password123');
    console.log('Logged in successfully');

    // Get user profile
    const userProfile = await app.getUserProfile('user123');
    console.log('User profile:', userProfile);

    // Get products
    const products = await app.getProducts({ category: 'electronics' });
    console.log('Products:', products);

    // Get specific product
    const product = await app.getProductDetails('product456');
    console.log('Product details:', product);

    // Logout
    app.logout();
    console.log('Logged out successfully');
```

```
  } catch (error) {
    console.error('Error:', error);
  }
}

// In a real application, we would call appDemo() to run the code
// For this example, we're showing the facade implementation
```

Composite Pattern

Concept: Composes objects into tree structures to represent part-whole hierarchies. Allows clients to treat individual objects and compositions of objects uniformly.

Key JavaScript Concepts:

- Tree structures
- Recursive operations
- Component interfaces
- Node hierarchies

Example 1: File System Structure

```
// Component interface (abstract)
class FileSystemComponent {
  constructor(name) {
    this.name = name;
  }

  // Methods to be implemented by concrete classes
  getSize() {
    throw new Error('Method getSize() must be implemented');
  }

  print(indent = 0) {
    throw new Error('Method print() must be implemented');
  }
}

// Leaf node - represents a file
class File extends FileSystemComponent {
  constructor(name, size) {
    super(name);
    this.size = size; // Size in KB
  }

  getSize() {
    return this.size;
  }

  print(indent = 0) {
    console.log(`${' '.repeat(indent)}File: ${this.name} (${this.size} KB)`);
  }
}

// Composite node - represents a directory that can contain files and other directories
class Directory extends FileSystemComponent {
  constructor(name) {
    super(name);
    this.children = [];
  }

  // Add a file or directory to this directory
  add(component) {
    this.children.push(component);
    return this; // Allow chaining
```

```
  }

  // Remove a component from this directory
  remove(component) {
    const index = this.children.indexOf(component);
    if (index !== -1) {
      this.children.splice(index, 1);
    }
    return this; // Allow chaining
  }

  // Get the total size of this directory (sum of all children)
  getSize() {
    return this.children.reduce((sum, child) => sum + child.getSize(), 0);
  }

  // Print the directory structure
  print(indent = 0) {
    console.log(`${' '.repeat(indent)}Directory: ${this.name} (${this.getSize()} KB)`);
    this.children.forEach(child => {
      child.print(indent + 2);
    });
  }
}

// Client code
// Create the file system structure
const root = new Directory('root');

const docs = new Directory('documents');
docs.add(new File('resume.pdf', 1000))
    .add(new File('cover-letter.docx', 350));

const pics = new Directory('pictures');
pics.add(new File('vacation.jpg', 3000))
    .add(new File('family.jpg', 2000));

const music = new Directory('music');
music.add(new File('song1.mp3', 4000))
     .add(new File('song2.mp3', 3500));

const work = new Directory('work');
work.add(new File('project.ppt', 2200))
```

```
    .add(new File('report.xlsx', 1800));

// Build the directory structure
root.add(docs)
    .add(pics);

docs.add(work);
root.add(music);

// Print the file system
root.print();
// Directory: root (18850 KB)
//   Directory: documents (5350 KB)
//     File: resume.pdf (1000 KB)
//     File: cover-letter.docx (350 KB)
//     Directory: work (4000 KB)
//       File: project.ppt (2200 KB)
//       File: report.xlsx (1800 KB)
//   Directory: pictures (5000 KB)
//     File: vacation.jpg (3000 KB)
//     File: family.jpg (2000 KB)
//   Directory: music (7500 KB)
//     File: song1.mp3 (4000 KB)
//     File: song2.mp3 (3500 KB)

// Get the total size
console.log(`Total size: ${root.getSize()} KB`);
// "Total size: 18850 KB"
```

Example 2: UI Component Composition

```
// Base Component class
class UIComponent {
  constructor(id) {
    this.id = id;
    this.parent = null;
  }

  setParent(parent) {
    this.parent = parent;
  }

  getParent() {
    return this.parent;
  }

  // Abstract methods
  render() {
    throw new Error('Method render() must be implemented');
  }

  getMarkup() {
    throw new Error('Method getMarkup() must be implemented');
  }
}

// Leaf component - Button
class Button extends UIComponent {
  constructor(id, text, onClick) {
    super(id);
    this.text = text;
    this.onClick = onClick;
  }

  render() {
    console.log(`Rendering Button: ${this.id} with text: ${this.text}`);
    // Actual rendering code would go here in a real implementation
  }

  getMarkup() {
    return `<button id="${this.id}" onclick="${this.onClick}">${this.text}</button>`;
  }
}
```

```
// Leaf component - Input
class Input extends UIComponent {
  constructor(id, type, placeholder) {
    super(id);
    this.type = type;
    this.placeholder = placeholder;
  }

  render() {
    console.log(`Rendering Input: ${this.id} of type: ${this.type}`);
    // Actual rendering code would go here in a real implementation
  }

  getMarkup() {
    return `<input id="${this.id}" type="${this.type}" placeholder="${this.placeholder}
  }
}

// Composite component - Container that can hold other components
class Container extends UIComponent {
  constructor(id, className = '') {
    super(id);
    this.className = className;
    this.children = [];
  }

  add(component) {
    this.children.push(component);
    component.setParent(this);
    return this; // Allow chaining
  }

  remove(component) {
    const index = this.children.indexOf(component);
    if (index !== -1) {
      this.children[index].setParent(null);
      this.children.splice(index, 1);
    }
    return this; // Allow chaining
  }

  render() {
    console.log(`Rendering Container: ${this.id} with class: ${this.className}`);
```

```
    // Render all children recursively
    this.children.forEach(child => child.render());
  }

  getMarkup() {
    // Combine markup of all children
    const childrenMarkup = this.children.map(child => child.getMarkup()).join('\n');
    return `<div id="${this.id}" class="${this.className}">\n${childrenMarkup}\n</div>`
  }
}

// Form component (another composite)
class Form extends Container {
  constructor(id, action, method = 'POST') {
    super(id);
    this.action = action;
    this.method = method;
  }

  render() {
    console.log(`Rendering Form: ${this.id} with action: ${this.action}`);
    // Render all children recursively
    this.children.forEach(child => child.render());
  }

  getMarkup() {
    // Combine markup of all children
    const childrenMarkup = this.children.map(child => child.getMarkup()).join('\n');
    return `<form id="${this.id}" action="${this.action}" method="${this.method}">\n${c
  }
}

// Client code
// Create a form with inputs and a button
const loginForm = new Form('loginForm', '/api/login');

const usernameInput = new Input('username', 'text', 'Enter username');
const passwordInput = new Input('password', 'password', 'Enter password');
const submitButton = new Button('submitBtn', 'Login', 'submitForm()');

// Add inputs to a container for layout purposes
const inputContainer = new Container('inputContainer', 'form-inputs');
inputContainer.add(usernameInput).add(passwordInput);
```

```
// Add the container and button to the form
loginForm.add(inputContainer).add(submitButton);

// Render the entire UI tree
loginForm.render();
// Output:
// Rendering Form: loginForm with action: /api/login
// Rendering Container: inputContainer with class: form-inputs
// Rendering Input: username of type: text
// Rendering Input: password of type: password
// Rendering Button: submitBtn with text: Login

// Get the HTML markup
console.log(loginForm.getMarkup());
// Output:
// <form id="loginForm" action="/api/login" method="POST">
// <div id="inputContainer" class="form-inputs">
// <input id="username" type="text" placeholder="Enter username" />
// <input id="password" type="password" placeholder="Enter password" />
// </div>
// <button id="submitBtn" onclick="submitForm()">Login</button>
// </form>
```

Proxy Pattern

Concept: Provides a surrogate or placeholder for another object to control access to it. Useful for lazy loading, access control, logging, and more.

Key JavaScript Concepts:

- Object access control
- Method interception
- Lazy initialization
- Same interface as the real object

Example 1: Image Proxy for Lazy Loading

```
// RealImage class that loads an image from a file
class RealImage {
  constructor(filename) {
    this.filename = filename;
    this.loadFromDisk();
  }

  loadFromDisk() {
    console.log(`Loading image: ${this.filename} from disk`);
    // In a real app, this would actually load the image
    // Simulate a delay
    return new Promise(resolve => setTimeout(resolve, 1000));
  }

  display() {
    console.log(`Displaying image: ${this.filename}`);
  }

  getFilename() {
    return this.filename;
  }
}

// ImageProxy that defers loading until necessary
class ImageProxy {
  constructor(filename) {
    this.filename = filename;
    this.realImage = null;
  }

  // Lazy load - only load when needed
  async display() {
    if (this.realImage === null) {
      console.log(`[Proxy] First-time display request for ${this.filename}`);
      this.realImage = new RealImage(this.filename);
      await this.realImage.loadFromDisk();
    } else {
      console.log(`[Proxy] Using cached image for ${this.filename}`);
    }

    this.realImage.display();
  }
```

```
  getFilename() {
    return this.filename;
  }
}

// Client code
async function runImageGallery() {
  console.log("Starting image gallery...");

  // Create proxy objects - images aren't loaded yet
  const image1 = new ImageProxy("image1.jpg");
  const image2 = new ImageProxy("image2.jpg");
  const image3 = new ImageProxy("image3.jpg");

  console.log("Image gallery created but no images loaded yet");
  console.log("------------------------------");

  // Display image1 - this will load it
  console.log("User clicks on image1:");
  await image1.display();
  console.log("------------------------------");

  // Display image2 - this will load it
  console.log("User clicks on image2:");
  await image2.display();
  console.log("------------------------------");

  // Display image1 again - this will use cached version
  console.log("User returns to image1:");
  await image1.display();
  console.log("------------------------------");

  // Image3 is never displayed, so it's never loaded
  console.log(`Image ${image3.getFilename()} was never displayed, so it was never loade
}

// Run the gallery demo
runImageGallery();
```

Example 2: Protection Proxy for Access Control

```
// Subject interface: Document system
class DocumentSystem {
  constructor() {
    this.documents = {
      'public': ['company_overview.pdf', 'product_catalog.pdf'],
      'confidential': ['financial_report.pdf', 'strategic_plan.pdf'],
      'restricted': ['employee_salaries.pdf', 'acquisition_plans.pdf', 'board_minutes.p
    };
  }

  getDocumentList(level) {
    return this.documents[level] || [];
  }

  getDocument(name) {
    // In a real system, this would return the actual document
    let docLevel = null;

    // Find which level the document belongs to
    for (const level in this.documents) {
      if (this.documents[level].includes(name)) {
        docLevel = level;
        break;
      }
    }

    if (docLevel) {
      return `Content of ${name} (${docLevel} level)`;
    } else {
      throw new Error(`Document not found: ${name}`);
    }
  }
}

// Protection Proxy for access control
class DocumentSystemProxy {
  constructor(user) {
    this.documentSystem = new DocumentSystem();
    this.user = user;

    // Access control rules
    this.accessLevels = {
      'guest': ['public'],
```

```
    'employee': ['public', 'confidential'],
    'manager': ['public', 'confidential', 'restricted']
  };
}

// Helper method to check access
hasAccess(level) {
  const allowedLevels = this.accessLevels[this.user.role] || [];
  return allowedLevels.includes(level);
}

// Get list of documents based on user permissions
getDocumentList() {
  const result = [];

  // Only return documents the user has access to
  for (const level in this.documentSystem.documents) {
    if (this.hasAccess(level)) {
      result.push(...this.documentSystem.documents[level]);
    }
  }

  return result;
}

// Get a specific document if the user has access
getDocument(name) {
  // Find which level the document belongs to
  let docLevel = null;

  for (const level in this.documentSystem.documents) {
    if (this.documentSystem.documents[level].includes(name)) {
      docLevel = level;
      break;
    }
  }

  if (!docLevel) {
    throw new Error(`Document not found: ${name}`);
  }

  // Check if user has access to this level
  if (!this.hasAccess(docLevel)) {
```

```
      console.log(`[SECURITY] Access denied: ${this.user.name} tried to access ${name}`
      throw new Error("Access denied. You don't have permission to access this document
    }

    // Log the access for audit
    console.log(`[AUDIT] ${this.user.name} (${this.user.role}) accessed ${name}`);

    // If they have access, get the document from the real system
    return this.documentSystem.getDocument(name);
  }
}

// Client code using the proxy
function documentSystemDemo() {
  // Create different users
  const guestUser = { name: 'John Visitor', role: 'guest' };
  const employeeUser = { name: 'Alice Worker', role: 'employee' };
  const managerUser = { name: 'Bob Director', role: 'manager' };

  // Create proxies for each user
  const guestProxy = new DocumentSystemProxy(guestUser);
  const employeeProxy = new DocumentSystemProxy(employeeUser);
  const managerProxy = new DocumentSystemProxy(managerUser);

  // Test with guest user
  console.log(`\nAvailable documents for ${guestUser.name}:`);
  console.log(guestProxy.getDocumentList());

  try {
    console.log(guestProxy.getDocument('company_overview.pdf'));
    console.log(guestProxy.getDocument('financial_report.pdf')); // Should fail
  } catch (error) {
    console.log(`Error: ${error.message}`);
  }

  // Test with employee user
  console.log(`\nAvailable documents for ${employeeUser.name}:`);
  console.log(employeeProxy.getDocumentList());

  try {
    console.log(employeeProxy.getDocument('financial_report.pdf'));
    console.log(employeeProxy.getDocument('acquisition_plans.pdf')); // Should fail
  } catch (error) {
```

```
    console.log(`Error: ${error.message}`);
  }

  // Test with manager user
  console.log(`\nAvailable documents for ${managerUser.name}:`);
  console.log(managerProxy.getDocumentList());

  try {
    console.log(managerProxy.getDocument('strategic_plan.pdf'));
    console.log(managerProxy.getDocument('acquisition_plans.pdf')); // Should work
  } catch (error) {
    console.log(`Error: ${error.message}`);
  }
}

documentSystemDemo();
```

I'll continue our design patterns discussion, focusing on the remaining behavioral patterns and other important patterns you've outlined.

Behavioral Patterns

Behavioral patterns focus on communication between objects, how they operate together, and how responsibilities are assigned among them.

Observer Pattern

The Observer pattern defines a one-to-many dependency between objects so that when one object (the subject) changes state, all its dependents (observers) are notified and updated automatically.

Key components:

- **Subject**: Maintains a list of observers and notifies them of state changes
- **Observer**: Provides an interface for objects that need to be notified
- **ConcreteSubject**: Stores state and sends notifications to observers
- **ConcreteObserver**: Implements the Observer interface to react to subject updates

Example in JavaScript:

```
// Subject (Observable)
class NewsPublisher {
  constructor() {
    this.subscribers = [];
    this.latestNews = null;
  }

  subscribe(observer) {
    this.subscribers.push(observer);
  }

  unsubscribe(observer) {
    this.subscribers = this.subscribers.filter(sub => sub !== observer);
  }

  publishNews(news) {
    this.latestNews = news;
    this.notifySubscribers();
  }

  notifySubscribers() {
    this.subscribers.forEach(observer => observer.update(this.latestNews));
  }
}

// Observer interface
class Subscriber {
  update(news) {
    // Abstract method to be implemented by concrete observers
  }
}

// Concrete Observer implementations
class EmailSubscriber extends Subscriber {
  constructor(email) {
    super();
    this.email = email;
  }

  update(news) {
    console.log(`Sending news to ${this.email}: ${news}`);
  }
}
```

```
class AppSubscriber extends Subscriber {
  constructor(userId) {
    super();
    this.userId = userId;
  }

  update(news) {
    console.log(`Sending push notification to User ${this.userId}: ${news}`);
  }
}

// Usage
const publisher = new NewsPublisher();
const emailSub = new EmailSubscriber("john@example.com");
const appSub = new AppSubscriber(12345);

publisher.subscribe(emailSub);
publisher.subscribe(appSub);

publisher.publishNews("New JavaScript framework released!");
// Output:
// Sending news to john@example.com: New JavaScript framework released!
// Sending push notification to User 12345: New JavaScript framework released!
```

Use cases:

- Event handling systems
- Subscription services
- MVC architecture (model notifies views)
- Reactive programming

Interview tips:

- Know how to implement both push (subject provides data) and pull (observer requests data) models
- Be able to discuss potential memory leaks if observers aren't properly unsubscribed
- Explain how this pattern helps maintain loose coupling

Mediator Pattern

The Mediator pattern reduces chaotic dependencies between objects by making them communicate indirectly through a mediator object.

Key components:

- **Mediator**: Defines an interface for communicating with colleague objects
- **ConcreteMediator**: Implements cooperative behavior by coordinating colleagues
- **Colleague**: Each colleague knows its mediator and communicates with other colleagues through it

Example in JavaScript:

```
// Mediator
class ChatRoom {
  constructor() {
    this.users = {};
  }

  register(user) {
    this.users[user.name] = user;
    user.chatroom = this;
  }

  send(message, from, to) {
    if (to) {
      // Direct message to specific user
      this.users[to].receive(message, from);
    } else {
      // Broadcast to all users except sender
      for (const key in this.users) {
        if (key !== from) {
          this.users[key].receive(message, from);
        }
      }
    }
  }
}

// Colleague
class User {
  constructor(name) {
    this.name = name;
    this.chatroom = null;
  }

  send(message, to) {
    this.chatroom.send(message, this.name, to);
  }

  receive(message, from) {
    console.log(`${this.name} received from ${from}: ${message}`);
  }
}

// Usage
```

```
const chatroom = new ChatRoom();

const john = new User("John");
const jane = new User("Jane");
const dave = new User("Dave");

chatroom.register(john);
chatroom.register(jane);
chatroom.register(dave);

john.send("Hello everyone!");
jane.send("Hey John", "John");
```

Use cases:

- Chat applications
- Air traffic control systems
- Complex form validation
- Coordinating UI components

Interview tips:

- Contrast with Observer pattern - mediator centralizes communication logic while observer distributes it
- Highlight how it helps maintain the Single Responsibility Principle
- Discuss potential performance bottlenecks if the mediator becomes too complex

Command Pattern

The Command pattern encapsulates a request as an object, allowing you to parameterize clients with different requests, queue requests, log them, and support undoable operations.

Key components:

- **Command**: Declares an interface for executing operations
- **ConcreteCommand**: Implements the Command interface by binding together an action and a receiver
- **Invoker**: Asks the command to execute the request
- **Receiver**: Knows how to perform the operations associated with the command
- **Client**: Creates and configures concrete commands

Example in JavaScript:

```
// Receiver
class Light {
  turnOn() {
    console.log("Light is now ON");
  }

  turnOff() {
    console.log("Light is now OFF");
  }
}

// Command interface
class Command {
  execute() {}
  undo() {}
}

// Concrete Commands
class TurnOnCommand extends Command {
  constructor(light) {
    super();
    this.light = light;
  }

  execute() {
    this.light.turnOn();
  }

  undo() {
    this.light.turnOff();
  }
}

class TurnOffCommand extends Command {
  constructor(light) {
    super();
    this.light = light;
  }

  execute() {
    this.light.turnOff();
  }
```

```
  undo() {
    this.light.turnOn();
  }
}

// Invoker
class RemoteControl {
  constructor() {
    this.command = null;
    this.history = [];
  }

  setCommand(command) {
    this.command = command;
  }

  pressButton() {
    this.command.execute();
    this.history.push(this.command);
  }

  pressUndo() {
    const command = this.history.pop();
    if (command) {
      command.undo();
    }
  }
}

// Usage
const light = new Light();
const turnOn = new TurnOnCommand(light);
const turnOff = new TurnOffCommand(light);

const remote = new RemoteControl();

// Turn on the light
remote.setCommand(turnOn);
remote.pressButton();  // Output: Light is now ON

// Turn off the light
remote.setCommand(turnOff);
remote.pressButton();  // Output: Light is now OFF
```

```
// Undo last command
remote.pressUndo();    // Output: Light is now ON
```

Use cases:

- GUI buttons and menu items
- Macro recording
- Multi-level undo/redo
- Transaction management
- Job scheduling systems

Interview tips:

- Emphasize how it decouples the object that invokes the operation from the one that knows how to perform it
- Explain how command objects can be stored and manipulated like any other object
- Discuss real-world examples like database transactions or text editors

Iterator Pattern

The Iterator pattern provides a way to access elements of an aggregate object sequentially without exposing its underlying representation.

Key components:

- **Iterator**: Defines interface for accessing and traversing elements
- **ConcreteIterator**: Implements the Iterator interface
- **Aggregate**: Defines interface for creating an Iterator object
- **ConcreteAggregate**: Implements the Aggregate interface

Example in JavaScript:

```
// Iterator interface
class Iterator {
  hasNext() {}
  next() {}
}

// Concrete Iterator
class ArrayIterator extends Iterator {
  constructor(array) {
    super();
    this.array = array;
    this.index = 0;
  }

  hasNext() {
    return this.index < this.array.length;
  }

  next() {
    return this.hasNext() ? this.array[this.index++] : null;
  }
}

// Aggregate interface
class Collection {
  createIterator() {}
}

// Concrete Aggregate
class NumberCollection extends Collection {
  constructor() {
    super();
    this.numbers = [];
  }

  add(number) {
    this.numbers.push(number);
  }

  createIterator() {
    return new ArrayIterator(this.numbers);
  }
}
```

```
// Usage
const numbers = new NumberCollection();
numbers.add(1);
numbers.add(2);
numbers.add(3);
numbers.add(4);

const iterator = numbers.createIterator();

while (iterator.hasNext()) {
  console.log(iterator.next());
}
// Output: 1, 2, 3, 4
```

Modern JavaScript already provides iteration protocols:

```
// Implementing Symbol.iterator
class NumberCollection {
  constructor() {
    this.numbers = [];
  }

  add(number) {
    this.numbers.push(number);
  }

  // Make this class iterable
  [Symbol.iterator]() {
    let index = 0;
    const numbers = this.numbers;

    return {
      next: () => {
        if (index < numbers.length) {
          return { value: numbers[index++], done: false };
        } else {
          return { done: true };
        }
      }
    };
  }
}

// Usage with for...of loop
const numbers = new NumberCollection();
numbers.add(1);
numbers.add(2);
numbers.add(3);

for (const num of numbers) {
  console.log(num);
}
// Output: 1, 2, 3
```

Use cases:

- Traversing collections without exposing their structure
- Providing multiple traversal methods for a data structure

- Supporting polymorphic iteration

Interview tips:

- Know both classical implementation and the ES6 iteration protocols
- Discuss how it enables polymorphic iteration across various collection types
- Explain iterator variations like internal vs. external iterators

Strategy Pattern

The Strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. It lets the algorithm vary independently from clients that use it.

Key components:

- **Strategy**: Declares an interface common to all supported algorithms
- **ConcreteStrategy**: Implements the algorithm using the Strategy interface
- **Context**: Maintains a reference to a Strategy object and delegates to it

Example in JavaScript:

```
// Strategy interface (implicit in JavaScript)
// Concrete Strategies
class RegularPriceStrategy {
  calculatePrice(price) {
    return price;
  }
}

class DiscountPriceStrategy {
  constructor(discountPercent) {
    this.discountPercent = discountPercent;
  }

  calculatePrice(price) {
    return price * (1 - this.discountPercent / 100);
  }
}

class SalePriceStrategy {
  constructor(salePrice) {
    this.salePrice = salePrice;
  }

  calculatePrice(price) {
    return Math.min(price, this.salePrice);
  }
}

// Context
class ShoppingCart {
  constructor(pricingStrategy) {
    this.pricingStrategy = pricingStrategy;
    this.items = [];
  }

  addItem(item) {
    this.items.push(item);
  }

  setPricingStrategy(pricingStrategy) {
    this.pricingStrategy = pricingStrategy;
  }
```

```
  calculateTotal() {
    return this.items.reduce((total, item) => {
      return total + this.pricingStrategy.calculatePrice(item.price);
    }, 0);
  }
}

// Usage
const regularPricing = new RegularPriceStrategy();
const discountPricing = new DiscountPriceStrategy(20); // 20% discount
const salePricing = new SalePriceStrategy(5); // $5 maximum price

const cart = new ShoppingCart(regularPricing);

cart.addItem({ name: "Item 1", price: 10 });
cart.addItem({ name: "Item 2", price: 20 });

console.log("Regular price:", cart.calculateTotal()); // Output: 30

cart.setPricingStrategy(discountPricing);
console.log("With 20% discount:", cart.calculateTotal()); // Output: 24

cart.setPricingStrategy(salePricing);
console.log("With $5 sale price cap:", cart.calculateTotal()); // Output: 10
```

Use cases:

- Sorting algorithms
- Payment processing strategies
- Compression algorithms
- Authentication strategies
- Validation rules

Interview tips:

- Contrast with Template Method pattern - Strategy uses composition, Template Method uses inheritance
- Explain how Strategy enables runtime algorithm switching
- Discuss its relationship with dependency injection

Other Important Patterns

MVC Pattern

The Model-View-Controller (MVC) pattern separates an application into three main components: Model (data), View (user interface), and Controller (business logic).

Key components:

- **Model**: Represents data and business logic
- **View**: Displays the data to the user
- **Controller**: Handles user input and updates model and view accordingly

Example in JavaScript (simplified):

```
// Model
class UserModel {
  constructor() {
    this.users = [];
    this.currentId = 1;
    this.onUserListChanged = null;
  }

  addUser(name, email) {
    const user = {
      id: this.currentId++,
      name,
      email
    };
    this.users.push(user);

    if (this.onUserListChanged) {
      this.onUserListChanged(this.users);
    }

    return user;
  }

  deleteUser(id) {
    this.users = this.users.filter(user => user.id !== id);

    if (this.onUserListChanged) {
      this.onUserListChanged(this.users);
    }
  }

  bindUserListChanged(callback) {
    this.onUserListChanged = callback;
  }
}

// View
class UserView {
  constructor() {
    this.app = document.getElementById('app');
    this.title = this.createElement('h1', 'Users');
    this.form = this.createElement('form');
    this.nameInput = this.createElement('input');
```

```
  this.nameInput.type = 'text';
  this.nameInput.placeholder = 'Name';

  this.emailInput = this.createElement('input');
  this.emailInput.type = 'email';
  this.emailInput.placeholder = 'Email';

  this.submitButton = this.createElement('button', 'Add User');
  this.submitButton.type = 'submit';

  this.userList = this.createElement('ul', '');

  this.form.append(this.nameInput, this.emailInput, this.submitButton);
  this.app.append(this.title, this.form, this.userList);

  this.onSubmit = null;
  this.onDelete = null;

  this._initListeners();
}

_initListeners() {
  this.form.addEventListener('submit', event => {
    event.preventDefault();

    if (this.onSubmit) {
      this.onSubmit(this.nameInput.value, this.emailInput.value);
    }

    this.nameInput.value = '';
    this.emailInput.value = '';
  });
}

createElement(tag, textContent) {
  const element = document.createElement(tag);
  if (textContent) {
    element.textContent = textContent;
  }
  return element;
}

displayUsers(users) {
```

```
    while (this.userList.firstChild) {
      this.userList.removeChild(this.userList.firstChild);
    }

    if (users.length === 0) {
      const p = this.createElement('p', 'No users to display');
      this.userList.append(p);
    } else {
      users.forEach(user => {
        const li = this.createElement('li', `${user.name}: ${user.email}`);
        const deleteButton = this.createElement('button', 'Delete');

        deleteButton.addEventListener('click', () => {
          if (this.onDelete) {
            this.onDelete(user.id);
          }
        });

        li.append(deleteButton);
        this.userList.append(li);
      });
    }
  }

  bindAddUser(handler) {
    this.onSubmit = handler;
  }

  bindDeleteUser(handler) {
    this.onDelete = handler;
  }
}

// Controller
class UserController {
  constructor(model, view) {
    this.model = model;
    this.view = view;

    // Bind methods
    this.view.bindAddUser(this.handleAddUser.bind(this));
    this.view.bindDeleteUser(this.handleDeleteUser.bind(this));
    this.model.bindUserListChanged(this.onUserListChanged.bind(this));
```

```
    // Initial render
    this.onUserListChanged(this.model.users);
  }

  handleAddUser(name, email) {
    this.model.addUser(name, email);
  }

  handleDeleteUser(id) {
    this.model.deleteUser(id);
  }

  onUserListChanged(users) {
    this.view.displayUsers(users);
  }
}

// Usage
const app = new UserController(new UserModel(), new UserView());
```

Use cases:

- Web applications
- Desktop applications
- Mobile applications
- Any UI-based system with data manipulation

Interview tips:

- Understand variations like MVP (Model-View-Presenter) and MVVM (Model-View-ViewModel)
- Explain benefits: separation of concerns, testability, maintainability
- Discuss framework implementations (React, Angular, Vue)

Mixin Pattern

The Mixin pattern provides a way to add functionality to classes without inheritance, allowing for code reuse and composition.

Example in JavaScript:

```
// Mixins (reusable functionality)
const swimmingMixin = {
  swim() {
    console.log(`${this.name} is swimming.`);
  }
};

const flyingMixin = {
  fly() {
    console.log(`${this.name} is flying.`);
  }
};

const speakingMixin = {
  speak(phrase) {
    console.log(`${this.name} says: ${phrase}`);
  }
};

// Base class
class Animal {
  constructor(name) {
    this.name = name;
  }
}

// Using mixins with Object.assign
class Duck extends Animal {
  constructor(name) {
    super(name);
  }
}

// Apply mixins
Object.assign(Duck.prototype, swimmingMixin, flyingMixin, speakingMixin);

// Another approach using a utility function
function applyMixins(targetClass, ...mixins) {
  mixins.forEach(mixin => {
    Object.getOwnPropertyNames(mixin).forEach(name => {
      if (name !== 'constructor') {
        targetClass.prototype[name] = mixin[name];
      }
```

```
    });
  });
}

class Fish extends Animal {
  constructor(name) {
    super(name);
  }
}

applyMixins(Fish, swimmingMixin);

// Usage
const donald = new Duck("Donald");
donald.swim();  // Donald is swimming.
donald.fly();   // Donald is flying.
donald.speak("Quack!");  // Donald says: Quack!

const nemo = new Fish("Nemo");
nemo.swim();  // Nemo is swimming.
// nemo.fly();  // Would throw an error
```

ES6 class mixin implementation:

```
// Mixin factory functions
const Swimming = (superclass) => class extends superclass {
  swim() {
    console.log(`${this.name} is swimming.`);
  }
};

const Flying = (superclass) => class extends superclass {
  fly() {
    console.log(`${this.name} is flying.`);
  }
};

const Speaking = (superclass) => class extends superclass {
  speak(phrase) {
    console.log(`${this.name} says: ${phrase}`);
  }
};

// Base class
class Animal {
  constructor(name) {
    this.name = name;
  }
}

// Creating classes with mixins (composition)
class Duck extends Swimming(Flying(Speaking(Animal))) {
  constructor(name) {
    super(name);
  }
}

class Fish extends Swimming(Animal) {
  constructor(name) {
    super(name);
  }
}

// Usage
const donald = new Duck("Donald");
donald.swim();  // Donald is swimming.
donald.fly();   // Donald is flying.
```

```
donald.speak("Quack!");  // Donald says: Quack!

const nemo = new Fish("Nemo");
nemo.swim();  // Nemo is swimming.
// nemo.fly();  // Would throw an error
```

Use cases:

- Adding shared functionality across unrelated classes
- Creating reusable behavior modules
- Alternative to multiple inheritance
- Implementing traits or interfaces

Interview tips:

- Explain the difference between mixins and inheritance
- Discuss composability and the power of combining multiple mixins
- Be aware of potential naming collisions and how to resolve them

Flyweight Pattern

The Flyweight pattern minimizes memory usage by sharing as much data as possible with similar objects, using intrinsic (shared) and extrinsic (unique) states.

Key components:

- **Flyweight**: Interface through which flyweights can receive and act on extrinsic state
- **ConcreteFlyweight**: Implements the Flyweight interface and stores intrinsic state
- **FlyweightFactory**: Creates and manages flyweight objects

Example in JavaScript:

```
// Flyweight (Character)
class Character {
  constructor(char, font, size) {
    this.char = char;
    this.font = font;
    this.size = size;
  }

  display(x, y) {
    console.log(`Character ${this.char} at (${x}, ${y}) with font ${this.font} and size
  }
}

// Flyweight Factory
class CharacterFactory {
  constructor() {
    this.characters = {};
  }

  getCharacter(char, font, size) {
    // Create a unique key for storing/retrieving flyweight
    const key = `${char}-${font}-${size}`;

    // Return existing character if already created
    if (!this.characters[key]) {
      this.characters[key] = new Character(char, font, size);
      console.log(`Creating new character: ${key}`);
    } else {
      console.log(`Reusing character: ${key}`);
    }

    return this.characters[key];
  }

  getCount() {
    return Object.keys(this.characters).length;
  }
}

// Client code
class TextEditor {
  constructor() {
    this.chars = [];
```

```
    this.factory = new CharacterFactory();
  }

  write(text, font, size) {
    const characters = text.split('');

    characters.forEach((char, index) => {
      // Get flyweight character
      const character = this.factory.getCharacter(char, font, size);

      // Store position (extrinsic state) with reference to the character
      this.chars.push({
        character: character,
        x: index * 10,
        y: 20
      });
    });
  }

  render() {
    this.chars.forEach(item => {
      // Pass extrinsic state to the flyweight
      item.character.display(item.x, item.y);
    });
  }

  getCharacterCount() {
    return this.chars.length;
  }

  getFlyweightCount() {
    return this.factory.getCount();
  }
}

// Usage
const editor = new TextEditor();
editor.write("Hello World!", "Arial", 12);
editor.write("Hello Again!", "Arial", 12);

console.log(`Total characters: ${editor.getCharacterCount()}`);
console.log(`Unique flyweights: ${editor.getFlyweightCount()}`);
editor.render();
```

Use cases:

- Text editors and document processing
- Game development (trees, terrain, particles)
- Object pools in graphics applications
- Caching systems

Interview tips:

- Explain the difference between intrinsic (shared) and extrinsic (context-specific) state
- Discuss memory optimization benefits and when they become significant
- Address potential performance trade-offs (time vs. space complexity)

Interview Frequency by Companies

Pattern	Google	Amazon	Microsoft	Facebook	Apple	Netflix
Factory	High	High	Medium	Medium	High	Medium
Singleton	Medium	High	High	Low	Medium	Low
Observer	High	Medium	High	High	Medium	High
Strategy	Medium	High	Medium	Medium	Medium	Medium
Command	Low	Medium	High	Low	High	Low
Proxy	Medium	Low	Medium	High	Low	Medium
Decorator	Medium	Medium	Low	Medium	Medium	High
Iterator	Low	Medium	Medium	Low	Medium	Low
MVC	High	High	High	High	High	High
Module	High	High	Medium	High	Medium	High

Note: This frequency table is based on general industry trends and may vary over time.

These patterns provide powerful tools for solving common design problems in software development. Understanding when and how to apply them will significantly improve your code structure, maintainability, and flexibility.