# **Disclaimer:**

COPYRIGHT NOTICE

©All content, materials, and intellectual property contained within this document, including but not limited to text, images, diagrams, graphics, code, and any other original works, are protected by copyright laws and international treaties. This intellectual property is the sole and exclusive property of SRIKANTH TEKUMUDI

Unauthorized use, reproduction, distribution, modification, or transmission of any portion of this document, in any form or by any means, electronic or mechanical, without the prior written permission of SRIKANTH TEKUMUDI, is strictly prohibited and may result in severe civil and criminal penalties.

By accessing, viewing, or using this document, you acknowledge and agree to abide by all applicable copyright laws and the terms and conditions outlined herein. This copyright notice serves as a legal warning and reminder that the unauthorized use of this intellectual property is strictly prohibited and will be vigorously enforced.

Thank you for respecting the intellectual property rights of SRIKANTH TEKUMUDI

## **You can expect questions like explain common design patterns and sold principles.**

## **JS and React Patterns and Solid principles**

<https://javascriptpatterns.vercel.app/patterns>

<https://www.patterns.dev/vanilla/observer-pattern>

Design patterns are templates or solution schemes for solving common software development problems.

Design patterns are divided into three categories: creational patterns, structural patterns, and behavioral patterns.

Creational design patterns consist of a group of design patterns that are concerned with object creation.

Design patterns can be broadly categorized into three main types based on their purpose and structure: creational patterns, structural patterns, and behavioral patterns. Let's see how each of the provided design patterns falls into these categories:

### **Creational Patterns:**

Creational patterns focus on object creation mechanisms, dealing with the process of object instantiation. They provide various ways to create objects, hiding the details of instantiation, and making the system independent of how its objects are created, composed, and represented.

### **Structural Patterns:**

Structural design patterns are a category of software design patterns that focus on how classes and objects can be composed to form larger structures. These patterns provide a way to achieve flexibility and maintainability in your code by promoting loose coupling and code reusability.

### **Behavioral Patterns:**

Behavioral patterns focus on communication between objects and how responsibilities are distributed between them. They deal with the interaction and responsibility between classes or objects, defining how they collaborate to complete tasks and responsibilities.

### **Modular Pattern:**

* **Type**: Not a specific design pattern but rather a design principle.
* **Purpose**: Focuses on breaking down complex systems into smaller, self-contained modules that are easier to understand, maintain, and reuse.
* **Usage**: Used across all stages of software development to promote modularity, encapsulation, and separation of concerns.

These patterns play crucial roles in software design and architecture, helping developers address common challenges and promote maintainability, scalability, and flexibility in their applications.

### **singleton pattern**

* **Type**: Creational Pattern
* **Purpose**: Ensures that a class has only one instance and provides a global point of access to that instance.
* **Usage**: Used when exactly one object is needed to coordinate actions across the system.
* **Example**: Managing a connection pool, logging system, or configuration settings.

class CounterSingleton {

#count = 0; // Private property for the counter (using # for private)

constructor() {

if (!CounterSingleton.instance) {

CounterSingleton.instance = this;

}

return CounterSingleton.instance; // Always return the existing instance

}

increment() {

this.#count++;

}

decrement() {

this.#count--;

}

getCount() {

return this.#count;

}

static getInstance() {

return new CounterSingleton(); // Enforces using the constructor

}

}

// Usage

const myCounter = new CounterSingleton()

myCounter.increment();

myCounter.increment();

const myCounter2=new CounterSingleton()

console.log(myCounter2.getCount()) //output 2

console.log(myCounter.getCount()); // Output: 2

### **Proxy Pattern:**

* **Type**: Structural Pattern
* **Purpose**: Provides a surrogate or placeholder for another object to control access to it.
* **Usage**: Used to add a layer of indirection to support controlled access, logging, remote access, or caching for objects.
* **Example**: Implementing lazy initialization, access control, or logging for sensitive operations or remote resources.

const Person={

name:"john",

age:34,

email:"srikt@gmail.ock"

}

let personProxy=new Proxy(Person,{

get:(target,prop)=>{

console.log("intercept get",prop)

return target[prop]

},

set:(target,prop,value)=>{

console.log("intercepted in set",value)

target[prop]=value

return true

}

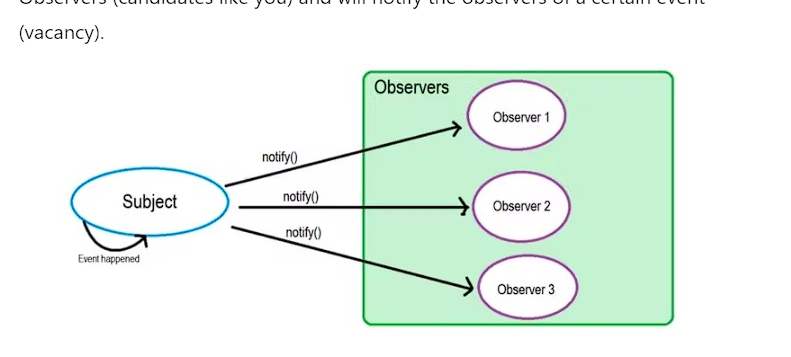
})

personProxy.name //intercept get name

personProxy.age="45" //intercepted in set 45

### **Observer Pattern:**

* **Type**: Behavioral Pattern
* **Purpose**: Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
* **Usage**: Used when objects need to be notified of changes in another object's state without tightly coupling the two.
* **Example**: Implementing event handling systems, UI components, or data synchronization mechanisms.



The observer pattern is a software design pattern in which an object, called the

subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods.

class EventObserver {

constructor() {

this.observers = [];

}

subscribe(fn) {

this.observers.push(fn);

}

unsubscribe(fn) {

this.observers.filter((subscriber) => subscriber != fn);

}

broadCast(data) {

this.observers.forEach((subscriber) => subscriber(data));

}

}

let observer = new EventObserver();

function logger(data) {

console.log(`logger 1 ${Date.now()} ${data}`);

}

function logger2(data) {

console.log(` logger 2 ${Date.now()} ${data}`);

}

observer.subscribe(logger);

observer.subscribe(logger2);

// observer.broadCast("update 1")

// observer.broadCast("update 2")

let inputEle = document.getElementById("input");

inputEle.addEventListener("input", (event) => {

let data = event.target.value;

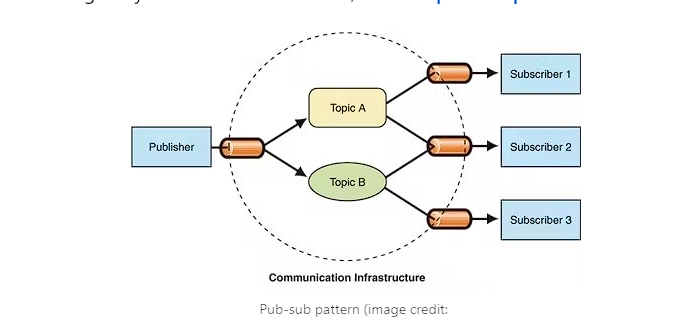
observer.broadCast(data);

});

<input type="text" id="input"/>

**Separation of Concerns**: The observer objects aren't tightly coupled to the observable object, and can be (de)coupled at any time. The observable object is responsible for monitoring the events, while the observers simply handle the received data.

### **publish subscribe pattern**

****

// PubSub implementation

class PubSub {

constructor() {

this.topics = {}; // Dictionary to store topics and their subscribers

}

// Method to subscribe to a topic

subscribe(topic, subscriber) {

if (!this.topics[topic]) {

this.topics[topic] = [];

}

this.topics[topic].push(subscriber);

}

// Method to unsubscribe from a topic

unsubscribe(topic, subscriber) {

if (this.topics[topic]) {

this.topics[topic] = this.topics[topic].filter(sub => sub !== subscriber);

}

}

// Method to publish a message to a topic

publish(topic, message) {

if (this.topics[topic]) {

this.topics[topic].forEach(subscriber => subscriber(message));

}

}

}

// Example usage:

const pubSub = new PubSub();

// Subscriber 1

const subscriber1 = message => {

console.log('Subscriber 1 received message:', message);

};

// Subscriber 2

const subscriber2 = message => {

console.log('Subscriber 2 received message:', message);

};

// Subscribe subscribers to a topic

pubSub.subscribe('news', subscriber1);

pubSub.subscribe('news', subscriber2);

// Publish a message to the 'news' topic

pubSub.publish('news', 'Breaking news: New JavaScript framework released!');

In the publisher-subscriber pattern, senders of messages, called *publishers*, do not program the messages to be sent directly to specific receivers, called *subscribers*.

This means that the publisher and subscriber don’t know about the existence of one another.

There is a third component, called *broker*, *message broker* or *event bus*, which is known by both the publisher and subscriber. It filters all incoming messages and distributes them accordingly.

* In the observer pattern, the observers are aware of the Subject. The Subject maintains a record of the Observers. Whereas, in publisher-subscriber, publishers and subscribers don’t need to know each other. They simply communicate with the help of message queues or a broker.
* In the publisher-subscriber pattern, components are loosely coupled as opposed to the observer pattern.
* The observer pattern is mostly implemented synchronously, i.e. the Subject calls the appropriate method of all its observers when an event occurs. The publisher-subscriber pattern is mostly implemented asynchronously (using a message queue).
* The observer pattern needs to be implemented in a single-application address space. On the other hand, the publisher-subscriber pattern is more of a cross-application pattern.

### **Factory Pattern:**

* **Type**: Creational Pattern
* **Purpose**: Defines an interface for creating objects, but subclasses decide which class to instantiate. It centralizes object creation logic, promoting flexibility and reusability.
* **Usage**: Used when the exact types of objects to be created are not known until runtime or when multiple related objects need to be created.
* **Example**: Creating different types of vehicles (car, truck, bus) in a manufacturing system.

1. **Definition**: The Factory pattern is a creational design pattern that provides a generic interface for creating objects. It centralizes object creation logic, promoting flexibility and reusability.
2. **Objective**: To encapsulate object creation, making it concise, reusable, and maintainable, especially in scenarios with dynamic object creation requirements.
3. **Implementation**:
   * Utilizes a factory function or class to create objects.
   * Centralizes object creation logic, eliminating redundant constructor calls.
   * Supports dynamic object creation based on conditions or configurations.
4. **Advantages**:
   * Promotes loose coupling by separating object creation from its implementation.
   * Enables dynamic creation of different object instances based on conditions.
   * Enhances code reusability and maintainability by encapsulating construction logic.
   * Exposes a defined interface to clients, abstracting the constructor or class.
5. **Disadvantages**:
   * Can introduce complexity, particularly in cases of intricate object creation logic.
   * Testing may be challenging due to the level of abstraction introduced by the pattern.
6. **Use Cases**:
   * Complex object creation processes involving dynamic factors.
   * Scenarios where different instances of an object need to be created based on conditions.
   * When a class cannot determine the subclass it must instantiate.
   * Creation of small objects sharing common properties.
7. **Comparison**:
   * **Factory Pattern vs Constructors**: Factory pattern enables dynamic object creation, while constructors instantiate objects with specific prototypes.
   * **Factory Pattern vs Singleton Pattern**: Factory pattern returns different instances based on conditions, whereas the singleton pattern ensures a single instance regardless of the object type.
8. **Conclusion**:
   * The Factory pattern is a powerful mechanism for object creation, promoting loose coupling and encapsulation.
   * It enhances code modularity, reusability, and maintainability.
   * Monitoring performance impact and user experience is essential after adopting the Factory pattern.

Certainly! Here are code examples demonstrating the Factory pattern in JavaScript:

### **Example 1: Basic Factory Function**

// Factory function to create car objects

function createCar(make, model, year) {

return {

make: make,

model: model,

year: year,

startEngine: function() {

console.log('Engine started for', this.make, this.model);

},

drive: function() {

console.log('Driving', this.make, this.model, this.year);

}

};

}

// Create car objects using the factory function

const car1 = createCar('Toyota', 'Camry', 2020);

const car2 = createCar('Honda', 'Civic', 2019);

// Test the created car objects

car1.startEngine();

car1.drive();

car2.startEngine();

car2.drive();

### **Example 2: Factory Class with Dynamic Object Creation**

// Factory class to create different types of vehicles

class VehicleFactory {

createVehicle(type) {

switch (type) {

case 'car':

return new Car();

case 'truck':

return new Truck();

case 'bus':

return new Bus();

default:

throw new Error('Invalid vehicle type: ' + type);

}

}

}

// Base class for vehicles

class Vehicle {

startEngine() {

console.log('Engine started for', this.constructor.name);

}

}

// Subclasses representing different types of vehicles

class Car extends Vehicle {

drive() {

console.log('Driving a car');

}

}

class Truck extends Vehicle {

drive() {

console.log('Driving a truck');

}

}

class Bus extends Vehicle {

drive() {

console.log('Driving a bus');

}

}

// Usage of the factory class

const factory = new VehicleFactory();

const car = factory.createVehicle('car');

const truck = factory.createVehicle('truck');

const bus = factory.createVehicle('bus');

car.startEngine();

car.drive();

truck.startEngine();

truck.drive();

bus.startEngine();

bus.drive();

These examples demonstrate how to use the Factory pattern to create objects dynamically and encapsulate object creation logic, promoting code reusability and maintainability.

Certainly! Let's discuss each of these patterns and how they align with design pattern categories:

### **Container-Presentation Pattern:**

* **Type**: Structural Pattern (in the context of UI architecture)
* **Purpose**: Separates concerns by dividing UI components into two categories: containers (smart components) and presentations (dumb components). Containers manage state and behavior, while presentations focus on rendering UI based on props.
* **Usage**: Used in UI development to improve code organization, maintainability, and reusability by decoupling UI logic from rendering.
* **Example**: React's container components (connected to Redux) act as containers, managing state and passing data to presentation components for rendering.

### **Render Prop Pattern:**

* **Type**: Behavioral Pattern (in the context of React component composition)
* **Purpose**: Allows components to share code by passing a render function as a prop, enabling flexible rendering behavior based on the component's state or logic.
* **Usage**: Used to share behavior between components, such as conditional rendering, data fetching, or event handling, without the need for higher-order components or component duplication.
* **Example**: A <Mouse> component that provides mouse position data to its children via a render prop function.

### **Provider Pattern:**

* **Type**: Creational or Behavioral Pattern (depending on the context)
* **Purpose**: Provides data or functionality to components down the component tree without explicit prop drilling, typically used for managing global state, theme, or other context-sensitive data.
* **Usage**: Used in frameworks like React to avoid prop drilling and provide access to shared resources across components.
* **Example**: React's <Provider> component from React Context API, which allows components to access shared state or functionality without passing props explicitly.

### **Hook Pattern:**

* **Type**: Behavioral Pattern (in the context of React functional components)
* **Purpose**: Enables functional components to use state and other React features without writing a class. Hooks provide a way to reuse stateful logic across multiple components without changing their structure.
* **Usage**: Used in React functional components to manage component state, side effects, and lifecycle methods, improving code organization and reusability.
* **Example**: useState, useEffect, and other built-in hooks provided by React for managing component state and lifecycle.

### **Composition Pattern:**

* **Type**: Structural Pattern
* **Purpose**: Combines multiple objects or components to create larger structures while maintaining flexibility and simplicity. Encourages composing complex systems from smaller, reusable parts rather than relying on inheritance or monolithic designs.
* **Usage**: Used in software development to build modular, scalable systems by composing smaller components, functions, or objects together.
* **Example**: Composing UI components in React by nesting smaller components within larger ones to create complex user interfaces.

### **Conclusion:**

Each of these patterns serves specific purposes in software development, promoting code organization, reusability, and maintainability. Whether used in UI development (Container-Presentation, Render Prop, Provider) or general software architecture (Hook, Composition), these patterns help developers build robust, flexible, and scalable applications by leveraging design principles such as separation of concerns, code reuse, and modularity. Understanding these patterns and their applicability can greatly improve the quality and architecture of software systems.

### **Creational Patterns:**

1. **Singleton Pattern**
2. **Factory Pattern**
3. **Prototype Pattern**

### **Structural Patterns:**

1. **Adapter Pattern**
2. **Composite Pattern**
3. **Proxy Pattern**

### **Behavioral Patterns:**

1. **Observer Pattern**

Additionally, let's categorize the remaining patterns:

### **UI Design Patterns (React):**

1. **Container-Presentation Pattern**
2. **Render Prop Pattern**
3. **Provider Pattern**
4. **Hook Pattern**
5. **Composition Pattern**

These categories help organize patterns based on their purposes and contexts of usage. Each pattern within a category addresses specific design challenges and provides solutions in a structured and reusable manner.

Solid principles

### **Explain Solid principles**

React's functional component approach and emphasis on props and state management naturally align well with several SOLID principles. Here's how you can integrate these principles effectively in your React applications:

**1. Single Responsibility Principle (SRP):**

* Break down complex components into smaller, well-defined functions. Each component should have a clear and focused responsibility, such as displaying a form, handling user input, or managing a specific section of the UI.
  + **Example:** Instead of a single ProductDetails component handling product information, reviews, and adding to cart functionality, create separate components like ProductInfo, ProductReviews, and AddToCart.

**2. Open/Closed Principle (OCP):**

* Leverage props and state to manage data flow and component behavior. This allows you to extend components without modifying the existing code.
  + **Example:** If you need to display a product in different contexts (e.g., product listing vs. product details page), add optional props like showDescription or showAddToCart to control the component's behavior.

**3. Liskov Substitution Principle (LSP):**

* While not as directly applicable as in object-oriented programming, strive for consistent and predictable behavior across components. When using a component in different contexts, ensure it functions as expected.
  + **Example:** Create a generic Button component that consistently receives an onClick prop for defining its action, regardless of the button's appearance or purpose (primary, secondary, etc.).

**4. Interface Segregation Principle (ISP):**

* Provide minimal and specific props to your components. Don't force them to receive data they don't need. Smaller, focused components are easier to reason about and reuse.
  + **Example:** If a ProductCard component only displays the product name and price, don't include unnecessary props like description or image URL unless they're relevant for all cards.

**5. Dependency Inversion Principle (DIP):**

* Depend on abstractions (functions or props) rather than concrete implementations. This promotes loose coupling and easier testing.
  + **Example:** Instead of hardcoding data fetching logic within a component, create a separate function (or use a data fetching library) and pass the fetched data as a prop to the component. This allows you to swap out the data fetching mechanism if needed.

**Additional Tips:**

* Use React's functional component approach to create reusable and stateless UI building blocks.
* Consider using higher-order components (HOCs) to encapsulate common functionality and share it across components.
* Leverage React's context API for managing application-wide state that needs to be shared by multiple components at different levels of the component hierarchy.
* Employ libraries like Redux or Zustand for complex state management needs, especially in larger applications.