Design Patterns for Interviews: In-Depth Guide with Examples

✅1. Singleton Pattern

Definition:

Ensures a class has only one instance and provides a global point of access to it.

Real-world Use Case:

Managing a single database connection across your app (e.g. MongoDB with Mongoose).

✔️Implementation (JS - MongoDB Mongoose Example)

```
// db.js
const mongoose = require("mongoose");

class Database {
  static instance;

  constructor() {
    if (Database.instance) return Database.instance;
    this.connected = false;
    Database.instance = this;
  }

  async connect(uri) {
    if (this.connected) return;
    await mongoose.connect(uri, {
      useNewUrlParser: true,
      useUnifiedTopology: true,
    });
    this.connected = true;
    console.log("✅ MongoDB Connected via Mongoose");
  }
}

module.exports = new Database();
```

✅Usage:

```
// index.js
const db = require("./db");
await db.connect("mongodb://localhost:27017/myapp");
```

All Mongoose models automatically use the default global connection established here.

✅2. Factory Pattern

Definition:

> Creates objects without exposing the instantiation logic to the client. Uses a factory method to return different class instances based on input.

✔️Basic Example

```
class Bike {
  drive() { console.log("Driving a Bike 🔴"); }
}

class Car {
  drive() { console.log("Driving a Car 🛢️"); }
}

class VehicleFactory {
  static create(type) {
    if (type === "bike") return new Bike();
    if (type === "car") return new Car();
    throw new Error("Unknown vehicle type");
  }
}

const vehicle = VehicleFactory.create("car");
vehicle.drive(); // Driving a Car 🛢️
```

☑️Real-world Factory Use Cases

1. Logger Factory

```
class ConsoleLogger {
  log(msg) { console.log(`[Console] ${msg}`); }
}

class FileLogger {
  log(msg) { console.log(`[File] Writing to file: ${msg}`); }
}

class DBLogger {
  log(msg) { console.log(`[DB] Inserting log: ${msg}`); }
}

class LoggerFactory {
  static getLogger(type) {
    switch (type) {
```

```
      case "dev": return new ConsoleLogger();
      case "prod": return new FileLogger();
      case "audit": return new DBLogger();
      default: throw new Error("Unknown logger type");
    }
  }
}

const logger = LoggerFactory.getLogger("dev");
logger.log("Hello world");
```

2. Payment Gateway Factory

```
class PayPalPayment {
  pay() { console.log("Paying via PayPal"); }
}

class RazorpayPayment {
  pay() { console.log("Paying via Razorpay"); }
}

class PaymentFactory {
  static getPayment(type) {
    if (type === 'paypal') return new PayPalPayment();
    if (type === 'razorpay') return new RazorpayPayment();
    throw new Error("Unsupported payment type");
  }
}

const payment = PaymentFactory.getPayment("razorpay");
payment.pay();
```

3. Notification Sender Factory

```
class EmailSender {
  send(msg) { console.log(`Sending Email: ${msg}`); }
}

class SMSSender {
  send(msg) { console.log(`Sending SMS: ${msg}`); }
}

class PushSender {
  send(msg) { console.log(`Sending Push Notification: ${msg}`); }
}

class NotificationFactory {
  static create(type) {
    switch (type) {
```

```
      case "email": return new EmailSender();
      case "sms": return new SMSSender();
      case "push": return new PushSender();
      default: throw new Error("Invalid notification type");
    }
  }
}

const notifier = NotificationFactory.create("sms");
notifier.send("User signed in");
```

✅3. Builder Pattern

Definition:

> Used to construct complex objects step-by-step. Useful when an object has many optional fields and you want to avoid long constructors.

Why Use It:

- Avoids telescoping constructors with too many parameters
- Cleaner object creation syntax with method chaining
- Makes code more readable and maintainable
- Only set the fields you care about; order doesn't matter
- Encapsulates construction logic inside the builder

Advantages:

- ✅Improves code readability and clarity
- ✅Supports optional parameters flexibly
- ✅Encourages immutability (when implemented accordingly)
- ✅Allows validations before creating the final object
- ✅Easy to add new optional fields

Disadvantages:

- ❌Slightly more verbose than using simple constructors
- ❌One extra class to maintain (the Builder)
- ❌Can be overkill for small, simple objects

✔️Example: Build a User Profile

```
class User {
  constructor(name, email, phone, address, isPremium) {
    this.name = name;
    this.email = email;
    this.phone = phone;
```

```
    this.address = address;
    this.isPremium = isPremium;
  }
}

class UserBuilder {
  constructor() {
    this.name = "";
    this.email = "";
    this.phone = "";
    this.address = "";
    this.isPremium = false;
  }

  setName(name) { this.name = name; return this; }
  setEmail(email) { this.email = email; return this; }
  setPhone(phone) { this.phone = phone; return this; }
  setAddress(address) { this.address = address; return this; }
  setIsPremium(isPremium) { this.isPremium = isPremium; return this; }

  build() {
    if (!this.name) throw new Error("Name is required");
    return new User(this.name, this.email, this.phone, this.address,
this.isPremium);
  }
}

const user = new UserBuilder()
  .setName("Sujith")
  .setEmail("sujith@example.com")
  .setIsPremium(true)
  .build();

console.log(user);
```

⚠️Interview Questions on Builder Pattern

1. **What is the Builder Pattern and when do you use it?**
2. **How is it different from Factory Pattern?**
3. **What are the advantages over constructors?**
4. **How to enforce required fields in builder?**
5. **Can you implement a real-world builder pattern for a user or order?**
6. **How would you validate fields before calling ``?**
7. **Drawbacks of Builder Pattern?**
8. **Can Factory and Builder work together?**

✅4. Strategy Pattern *(Behavioral Pattern)*

Definition:

> The Strategy Pattern defines a family of algorithms (behaviors), encapsulates each one, and makes them interchangeable at runtime.

Real-World Use Case (Static Strategy): Using different behavior for marking success/failure on different entity types (Bill, CreditNote, DebitNote) — each has a unique way of handling.

```
class Bill {
  markSuccess() { console.log("✅ Bill marked as paid and archived."); }
  markFailed() { console.log("❌ Bill marked as failed. Trigger retry."); }
}

class CreditNote {
  markSuccess() { console.log("✅ Credit Note applied successfully."); }
  markFailed() { console.log("❌ Failed to apply Credit Note."); }
}

class DBNote {
  markSuccess() { console.log("✅ Debit Note logged and emailed."); }
  markFailed() { console.log("❌ Debit Note action failed."); }
}

class EntityFactory {
  static create(type) {
    switch (type) {
      case "bill": return new Bill();
      case "cn": return new CreditNote();
      case "db": return new DBNote();
      default: throw new Error("Unknown entity type");
    }
  }
}

const entity = EntityFactory.create("cn");
entity.markSuccess();
entity.markFailed();
```

Dynamic Strategy Example:

```
class PayPalStrategy {
  pay(amount) { console.log(`Paying ₹${amount} via PayPal`); }
}

class RazorpayStrategy {
  pay(amount) { console.log(`Paying ₹${amount} via Razorpay`); }
}
```

```
class PaymentProcessor {
  constructor(strategy) {
    this.strategy = strategy;
  }
  setStrategy(strategy) {
    this.strategy = strategy;
  }
  pay(amount) {
    this.strategy.pay(amount);
  }
}

const processor = new PaymentProcessor(new PayPalStrategy());
processor.pay(500);
processor.setStrategy(new RazorpayStrategy());
processor.pay(300);
```

✅5. Observer Pattern *(Behavioral Pattern)*

Definition:

> An object (subject) maintains a list of dependents (observers) and notifies them automatically when its state changes.

✔️Custom Implementation: News Agency

```
class NewsAgency {
  constructor() {
    this.subscribers = [];
  }

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

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

  notify(news) {
    this.subscribers.forEach(sub => sub.update(news));
  }
}

class Reader {
  constructor(name) {
```

```
    this.name = name;
  }

  update(news) {
    console.log(`${this.name} received news: ${news}`);
  }
}

const agency = new NewsAgency();
const alice = new Reader("Alice");
const bob = new Reader("Bob");

agency.subscribe(alice);
agency.subscribe(bob);
agency.notify("Observer Pattern is awesome!");
```

✔️Node.js `EventEmitter` Example (Order System)

```
const EventEmitter = require('events');

class OrderService extends EventEmitter {
  createOrder(order) {
    console.log("✅ Order created", order);
    this.emit("order_created", order);
  }
}

const orderService = new OrderService();

orderService.on("order_created", (order) => {
  console.log("Ⓜ️ Sending email for order:", order.id);
});

orderService.on("order_created", (order) => {
  console.log("🈂️ Logging analytics for:", order.id);
});

orderService.createOrder({ id: 123, amount: 500 });
```

✔️Real-World Notification System Example

```
class NotificationService {
  constructor() {
    this.observers = [];
  }
```

```
  subscribe(observer) {
    this.observers.push(observer);
  }

  unsubscribe(observer) {
    this.observers = this.observers.filter(obs => obs !== observer);
  }

  notifyAll(message) {
    this.observers.forEach(observer => observer.notify(message));
  }
}

class EmailService {
  notify(msg) { console.log(`Ⓜ️ Email: ${msg}`); }
}

class SMSService {
  notify(msg) { console.log(`♾️ SMS: ${msg}`); }
}

class PushService {
  notify(msg) { console.log(`♆ Push: ${msg}`); }
}

const notificationSystem = new NotificationService();
notificationSystem.subscribe(new EmailService());
notificationSystem.subscribe(new SMSService());
notificationSystem.subscribe(new PushService());

notificationSystem.notifyAll("‼️ User signed up!");
```

⚠️Observer vs Pub/Sub

Feature	Observer Pattern	Pub/Sub Pattern
Coupling	Tightly Coupled (Observers know Subject)	Loosely Coupled (via Broker)
Communication	Direct function call	Broker delivers events
Awareness	Observer knows Subject	Publisher & Subscriber are unaware of each other
Example	JS classes, EventEmitter	Redis Pub/Sub, Kafka, RabbitMQ

⚠️Interview Questions on Observer Pattern

1. What is the Observer Pattern?

2. Difference between Observer and Pub/Sub?
3. Real use cases: NotificationService, Order Events, Audit Logging?
4. How to implement Observer in JS?
5. EventEmitter vs custom pattern?
6. When should you use Pub/Sub instead?
7. Can you decouple notification system using Observer?

We will continue to expand this document with more patterns:

-