Topic 5: Design Patterns (Singleton, Factory, Observer, Strategy with TypeScript Implementations)

1. Problem Statement

CoffeeShip Café's Growing Pains

CoffeeShip Café is booming, but challenges are brewing:

- Sometimes, two managers try to run the shop at once, causing confusion.
- Adding new drinks to the menu is a headache.
- When a drink is prepared, inventory, baristas, and the customer all need updates-if one is missed, chaos erupts.
- The café wants to offer seasonal promotions and different ways to prepare drinks, but changing the workflow is risky.

How can CoffeeShip Café keep operations smooth, reliable, and ready to grow-even as new drinks, staff, and features are added?

2. Learning Objectives

By the end of this lesson, you'll be able to:

- Recognize and implement Singleton, Factory, Observer, and Strategy design patterns in TypeScript.
- Apply these patterns to solve real-world scaling and coordination problems.
- Build systems that are modular, maintainable, and easy to extend.

3. Concept Introduction with Analogy

The Café Playbook Analogy

- **Singleton:** Only one head manager can run the café at any time.
- **Factory:** The kitchen has a recipe book for making any drink, so new drinks can be added easily.
- **Observer:** When a drink is ready, all relevant staff (inventory, barista, customer) are instantly notified-no one is left out.
- **Strategy:** The barista can use different preparation styles (fast, eco, strong) for each order, and can swap styles as needed.

What Are Design Patterns?

Design patterns are **repeatable solutions to common problems** that come up when designing software systems.
Think of them as “best practice templates” or “recipes” that experienced developers use to solve certain types of challenges.
They aren’t finished code, but rather **ways of organizing your program** so it’s easier to build, change, and understand.

The Four Patterns (with Simple Explanations & Analogies)

a. Singleton

- **What it is:**
  Ensures there is **only one instance** of a particular object throughout your entire application

- **Analogy:**
  Like a café with only one manager-no matter how many times you ask for the manager, you always get the same person.

- **Why use it:**
  To manage things that should only exist once, such as a settings manager, database connection, or logging service.
  Example:

```
class CafeManager {
 private static instance: CafeManager;
 private constructor() {}
 static getInstance(): CafeManager {
   if (!CafeManager.instance) {
     CafeManager.instance = new CafeManager();
   }
   return CafeManager.instance;
 }
}
// Usage:
const manager1 = CafeManager.getInstance();
const manager2 = CafeManager.getInstance();
console.log(manager1 === manager2); // true
```

- **Key point:**
  There’s always just one manager.

b. Factory

- **What it is:**
  Provides a way to **create objects** (like drinks, users, or shapes) without specifying the exact class/type to create.

- **Analogy:**
  Like a barista who can make any drink you order (“latte,” “espresso,” “tea”)-you just tell them what you want, not how to make it.

- **Why use it:**
  To make it easy to add new types of objects without changing lots of code.

- **Example:**

```
interface Drink { serve(): void; }
class Latte implements Drink { serve() { console.log("Latte!"); } }
class Espresso implements Drink { serve() { console.log("Espresso!"); } }
class DrinkFactory {
  static createDrink(type: string): Drink {
```

```
    if (type === "latte") return new Latte();
    if (type === "espresso") return new Espresso();
    throw new Error("Unknown drink");
  }
}
// Usage:
const drink = DrinkFactory.createDrink("latte");
drink.serve(); // "Latte!"
```

- **Key point:**
  You can add new drinks (types) without rewriting the whole menu.

c. Observer

- **What it is:**
  Lets one object (the “subject”) **notify many other objects** (the “observers”) when something changes.
- **Analogy:**
  Like a barista ringing a bell when a drink is ready-everyone who cares (inventory, customer, promotion system) hears the bell and reacts.
- **Why use it:**
  To keep different parts of your system in sync without tightly coupling them.
- **Example:**

```
interface Observer { update(msg: string): void; }
class Customer implements Observer { update(msg: string) { console.log("Customer:", msg); } }
class Inventory implements Observer { update(msg: string) { console.log("Inventory:", msg); } }
class DrinkOrder {
  private observers: Observer[] = [];
  addObserver(obs: Observer) { this.observers.push(obs); }
  notifyAll(msg: string) { this.observers.forEach(obs => obs.update(msg)); }
  completeOrder() { this.notifyAll("Order complete!"); }
}
// Usage:
const order = new DrinkOrder();
order.addObserver(new Customer());
order.addObserver(new Inventory());
order.completeOrder();
// Output: Customer: Order complete! Inventory: Order complete!
```

- **Key point:**
  When something important happens, everyone who needs to know is notified automatically.

4. Strategy

- **What it is:**
  Lets you **swap out different ways of doing something** (an “algorithm”) without changing the object that uses it.
- **Analogy:**
  Like a barista choosing how to prepare a drink: fast, eco-friendly, or extra strong-just swap the recipe card.
- **Why use it:**
  To change behavior on the fly, or to keep your code flexible and clean.
- **Example:**

```
interface PrepStrategy { prepare(): void; }
class FastPrep implements PrepStrategy { prepare() { console.log("Fast prep!"); } }
```

```
class EcoPrep implements PrepStrategy { prepare() { console.log("Eco prep!"); } }
class Barista {
  constructor(private strategy: PrepStrategy) {}
  setStrategy(strategy: PrepStrategy) { this.strategy = strategy; }
  makeDrink() { this.strategy.prepare(); }
}
// Usage:
const barista = new Barista(new FastPrep());
barista.makeDrink(); // "Fast prep!"
barista.setStrategy(new EcoPrep());
barista.makeDrink(); // "Eco prep!"
```

- **Key point:**
  You can switch how things are done without rewriting the barista.

Pattern	What It Solves	Analogy	When to Use
Singleton	Only one instance needed	One café manager	Global settings, logging, config
Factory	Flexible object creation	Barista makes any drink	Many types, easy to add new ones
Observer	Notify many when something changes	Bell rings, all react	Events, updates, notifications
Strategy	Swap algorithms/behaviors easily	Swap recipe cards	Change behavior at runtime

6. Challenge

Your Turn!

- Implement a `PromotionSystem` observer that announces special offers to customers when a drink is served.
- Add it to the `DrinkOrder` notification list and test it.

7. Quick Recap & Key Takeaways

- **Singleton:** Ensures only one manager is in charge.
- **Factory:** Makes adding new drinks simple and safe.
- **Observer:** Instantly updates all staff and customers about important events.
- **Strategy:** Lets baristas switch preparation methods as needed.

8. (Optional) Programmer's Workflow Checklist

- Need only one instance? Use Singleton.
- Want to create objects flexibly? Use Factory.
- Need to notify many parties? Use Observer.
- Want to swap behaviors easily? Use Strategy.

9. Coming up next:

Master **Dependency Injection**-the secret ingredient for building flexible, testable, and loosely coupled systems.
Imagine your café's staff and machines always getting the right tools, ingredients, and helpers-automatically!