Problem Statement

You started build a Duck Simulation App to model different duck types with unique behaviors (sounds, flying, swim). Initially, with only a couple of ducks (e.g., Indian, American), the design worked fine.

As the app grew, more ducks were added (e.g., Wooden, Rubber, etc) each with distinct combinations of behaviors-some can’t fly, some squeak, some only float. To handle this, the Duck class became with many if-else checks and behavior logic.

This lead to:

- Code duplication: Repeating the same behaviors across ducks.
- Rigid design: Adding new ducks required modifying the Duck class, violating the Open/Closed Principle.
- Collaboration issues: Frequent merge conflicts as multiple developers updated the same class.

Problem Statement

You are building an e-commerce platform that initially had Credit Card Payments. As the platform expanded, the team added support for PayPal, Strip, Bitcoin and Apple Pay.

Initially, all payment methods were implemented inside one class: Payment Processor. As new payment options were added, this class became:

- Large and difficult to maintain
- Prone to bugs when changes were made
- Hard to extend without modifying the original logic

Result:

- The team faced merge conflicts, slow reviews, and high maintenance effort.

Core Requirements

1. The system should support multiple payment methods
2. It should be easy to add new strategies without modifying existing code.
3. Maintainability and extensibility are top priorities

⇒ Strategy DP

1. Sorting
2. Duck simulation App.

   B → Swim, Fly, Sound.

   Ducks
3. Payment

   UPI, Card, crypto, etc.

Is it mandatory to use DPs in a system?

No so what we'll use ?

↓

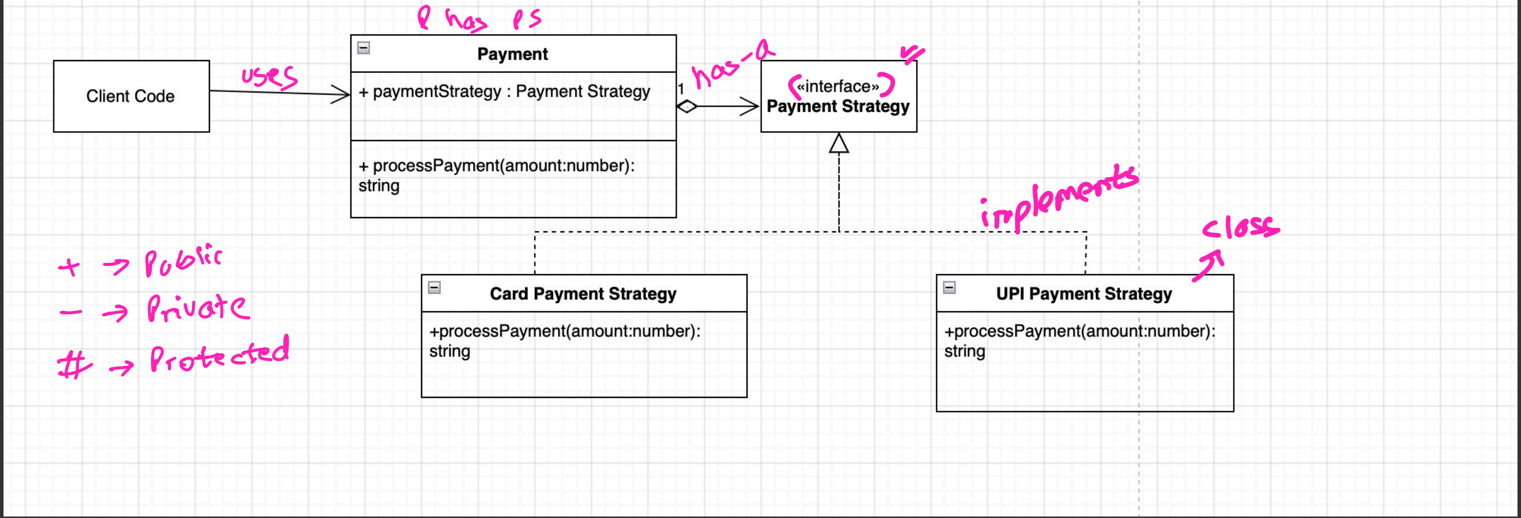
Basic (Fundamentals)

↓

OOPS + SOLID

Strategy Design Pattern

- Pre-defined Algorithms: Sorting, Discounting Logic, etc.
- Swithc between algorithms
- Reduce Code Duplication
- If 2-3 strategies, might be an overkill
- OCP
- Too many classes



Client Code

Payment

+ paymentStrategy : Payment Strategy

+ processPayment(amount:number): string

«interface»
Payment Strategy

Card Payment Strategy

+processPayment(amount:number): string

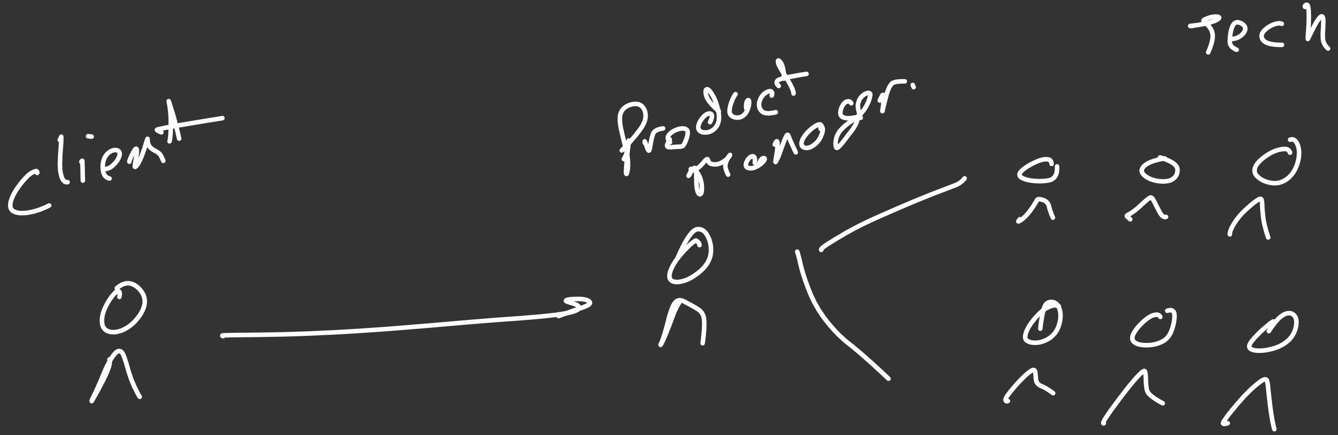
UPI Payment Strategy

+processPayment(amount:number): string

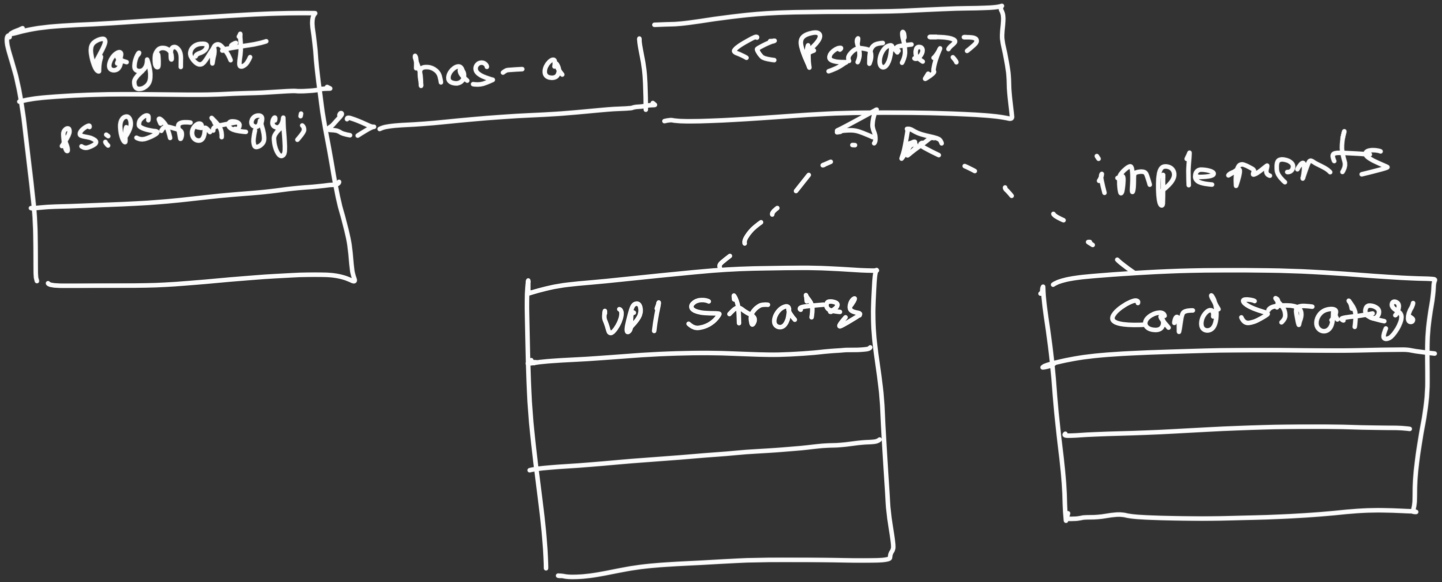
+ → Public

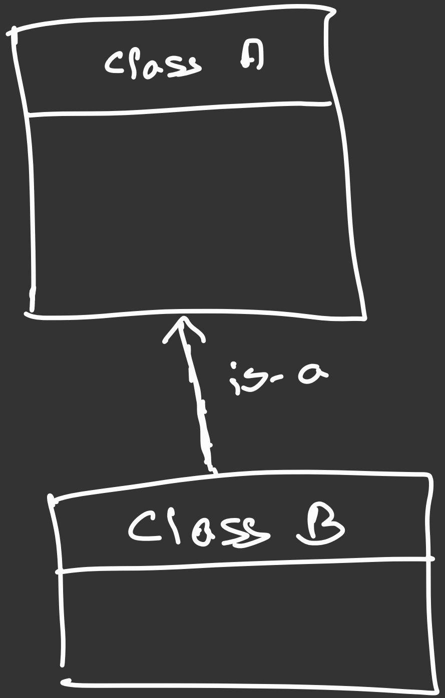
- → Private

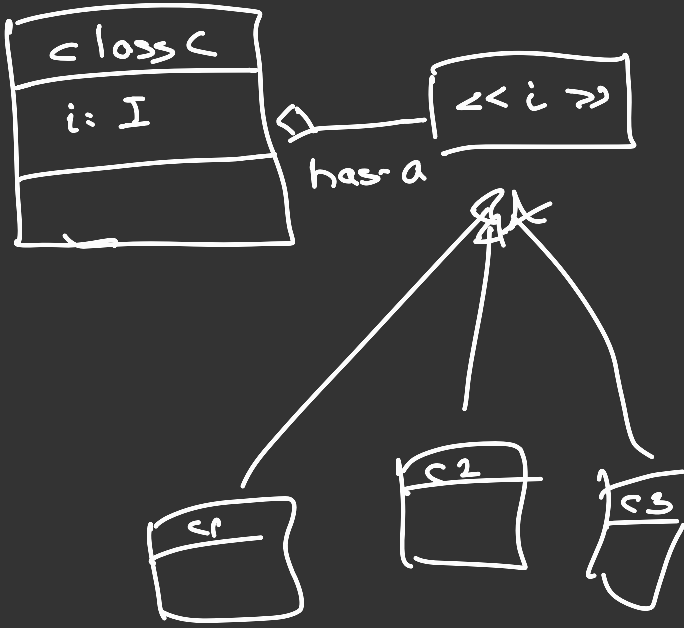
→ Protected


Client
Product managr.
Tech

Class Diagram of Strategy Design Pattern



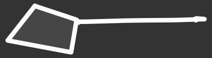




website: draw.io → UML

=> Has-a.

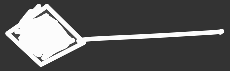
Aggregation and Composition



weak

```
class A {
    c: C
}
```

injected from outside



strong.

```
class B {
    c: C = new C();
}.
```

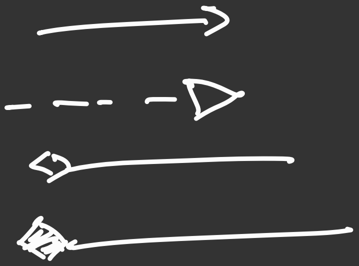
ownership.

```
class C
```

A

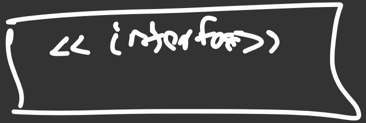
Class Diagram rules

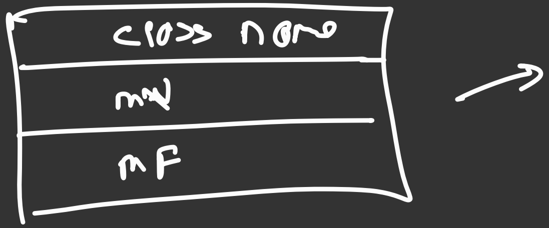
- Inheritance
- Interface
- Aggregation
- Composition



(Hollow Diav)

(solid —)





+ Public

- Private

Protected

Compilant

Viol^n

SOLID

↓

LSP

↓

inheritance (is-a)

↓

true inh

true is-a.

Media

→ A

→ V

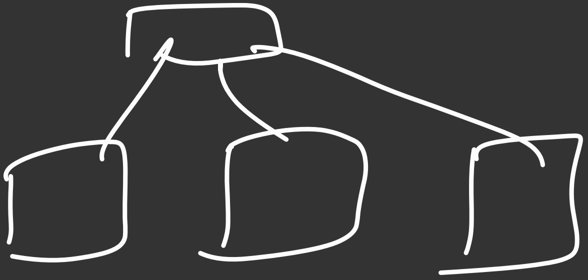
Audior ext M

→ A

→ V ✗

DIP

Active Recall



Strategy DP

- family of algos
- Payment
- Sorting.

Observer → Pub - Sub

Order

↳ Tracking

↳ Inventory

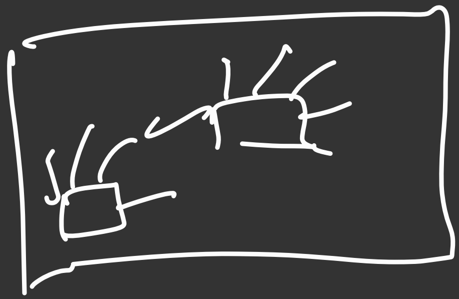
↳ Notific^{n}

Product

Out of Stock → Listed

↓

Notify.



Design Pattern

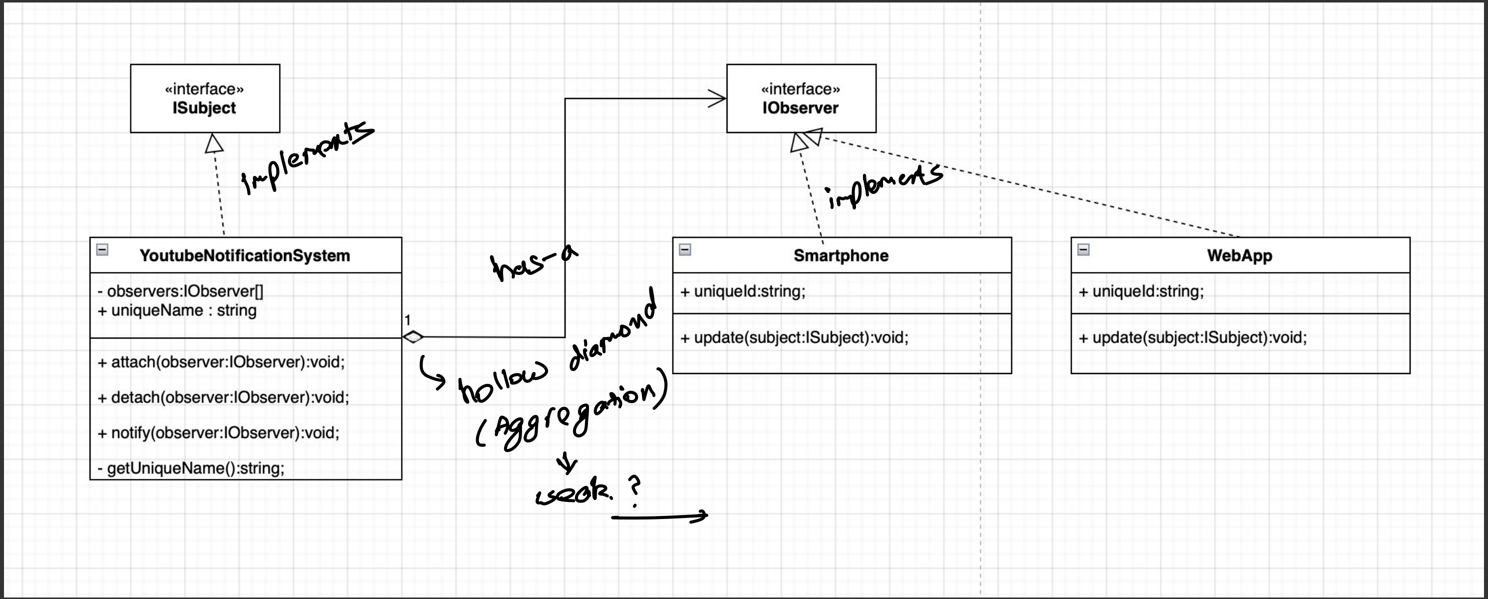
1. Problem
2. Sol^n → DP
3. Impl^n → Code
   → Class Diagram
4. Analytical : + , − , Applic^{n}s

Observer Design Pattern

Pub - Sub model

Problem Statement:

In the context of modern smart devices and IoT systems, multiple devices (such as smartphones, tablets, smartwatches, etc.) need to receive and react to updates from a central system, like a weather station. The weather station needs to notify all its observers (smart devices) whenever a weather update occurs. These observers should be able to react to the changes without being tightly coupled to the weather station, allowing for easy addition or removal of new devices.



«interface»
ISubject

«interface»
IObserver

YoutubeNotificationSystem

- observers:IObserver[]
+ uniqueName : string

+ attach(observer:IObserver):void;

+ detach(observer:IObserver):void;

+ notify(observer:IObserver):void;

- getUniqueName():string;

Smartphone

+ uniqueId:string;

+ update(subject:ISubject):void;

WebApp

+ uniqueId:string;

+ update(subject:ISubject):void;

⇒ Queries.

1) Stop working in RUST

2) Guidelines — Project.

Strategy ✓

Observer ✓

Decorator → ongoing

Singleton →

factory

Decorator Design Pattern

pst

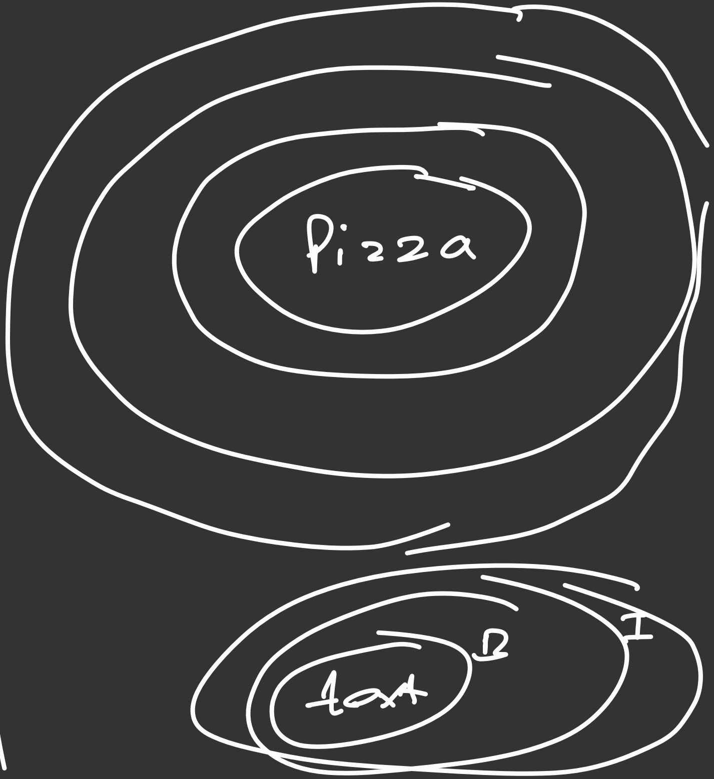
Problem: Customiz^n → explosion of classes

P D → oil
→ Ghee
→ Butter

M D → oil
→ Butter
→ Cheese

M R D → Paneer.
→ C + P + B

Pizza



```
                                              cheeseDec, {
     Pizza{

        getDesc();
        getPrice();


     }                                            }

  F   {                  M {
       getD   FH                   M
       getP   100                  200
       }                           }
```

Problem Statement: Pizza Ordering System using Decorator Pattern

You are tasked to implement a Pizza Ordering System where a customer can order different types of pizzas with multiple toppings. The goal is to dynamically add toppings to pizzas at runtime without creating a subclass for every possible combination.

Requirements:

Interface /

Abstract class

1. Base Pizza Classes:
   - Create an abstract class Pizza with methods:

     getDescription(): string → returns the name of the pizza.

     getPrice(): number → returns the base price of the pizza.

   - Implement at least two concrete pizzas: Margherita and Farmhouse.

2. Toppings as Decorators:
   - Create an abstract decorator class ToppingDecorator that extends Pizza.
   - Each topping class (e.g., Cheese, Olives, Jalapeno) should:

     ↓ ↓ ↓

     20 40 60

     Wrap a Pizza object.

     Override getDescription() to append its name.

     Override getPrice() to add its cost to the base pizza.

3 Dynamic Topping Addition:

- Allow users to create pizzas with multiple toppings by wrapping the pizza object with multiple decorator objects at runtime.
- Example: A Margherita with double cheese and jalapeno should be represented as:

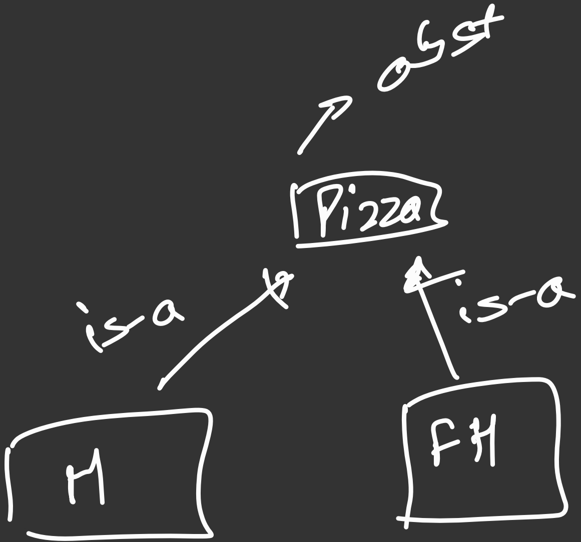
```
const myPizza = new Cheese(new Cheese(new Jalapeno(new Margherita())));
```

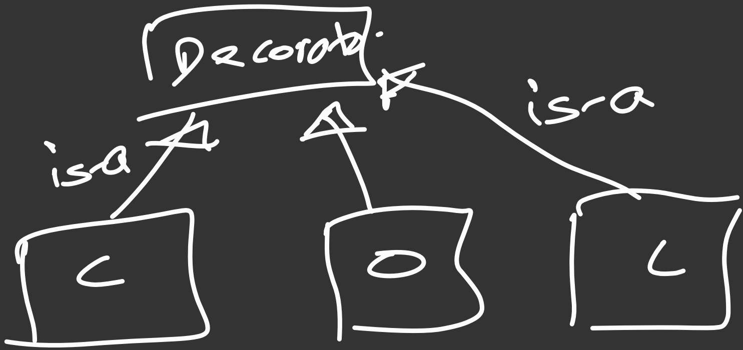
4 Output:

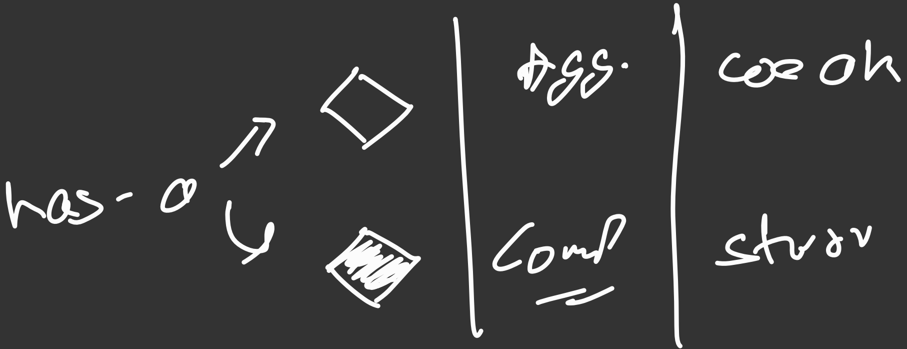
- getDescription() should return a string describing the pizza with all toppings.
- getPrice() should return the total cost including toppings.

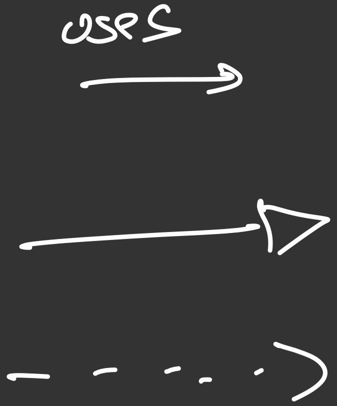
5 Constraints & Concepts:

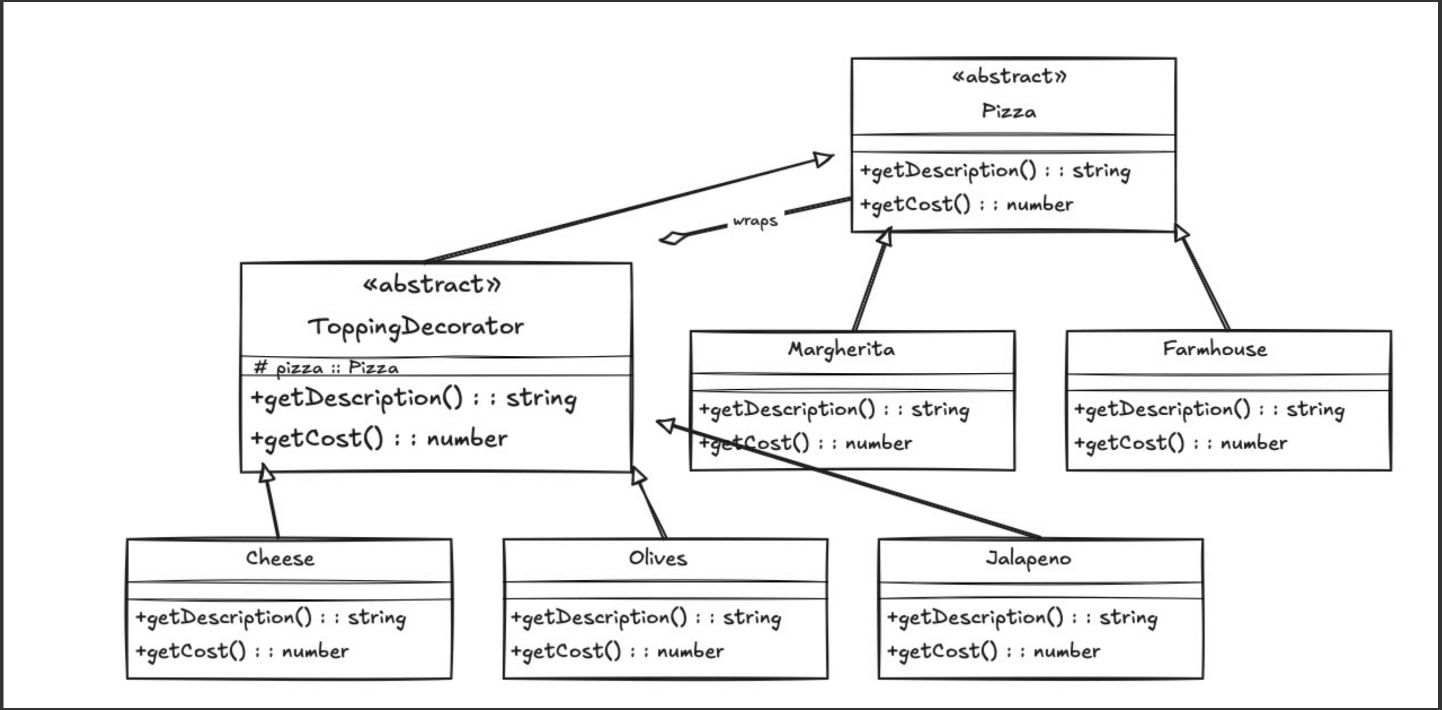
- Toppings are decorators — they extend pizza dynamically at runtime.
- Use protected pizza in the decorator to allow access only in the decorator and its subclasses.
- Call super() in the decorator constructor before assigning the pizza.


object
Pizza
is-a
is-a
M
FH


Decorator
is-a
is-a
C
O
C


has-a
Agg.
Comp
cor ok
strov


uses


«abstract»
Pizza
+getDescription() : : string
+getCost() : : number
wraps
«abstract»
ToppingDecorator
# pizza :: Pizza
+getDescription() : : string
+getCost() : : number
Margherita
+getDescription() : : string
+getCost() : : number
Farmhouse
+getDescription() : : string
+getCost() : : number
Cheese
+getDescription() : : string
+getCost() : : number
Olives
+getDescription() : : string
+getCost() : : number
Jalapeno
+getDescription() : : string
+getCost() : : number

Problem Statement 2:

In a cafe, customers can order various beverages, such as tea and coffee. These beverages can have different combinations of ingredients like honey, sugar, and whipped cream. The system needs to handle the following:

Each beverage has a description and a cost.

Beverages can be customised with additional ingredients by applying decorators to the basic beverage.

The decorators should allow the system to dynamically alter the beverage's cost based on the ingredients added, while still adhering to the object-oriented design principles.

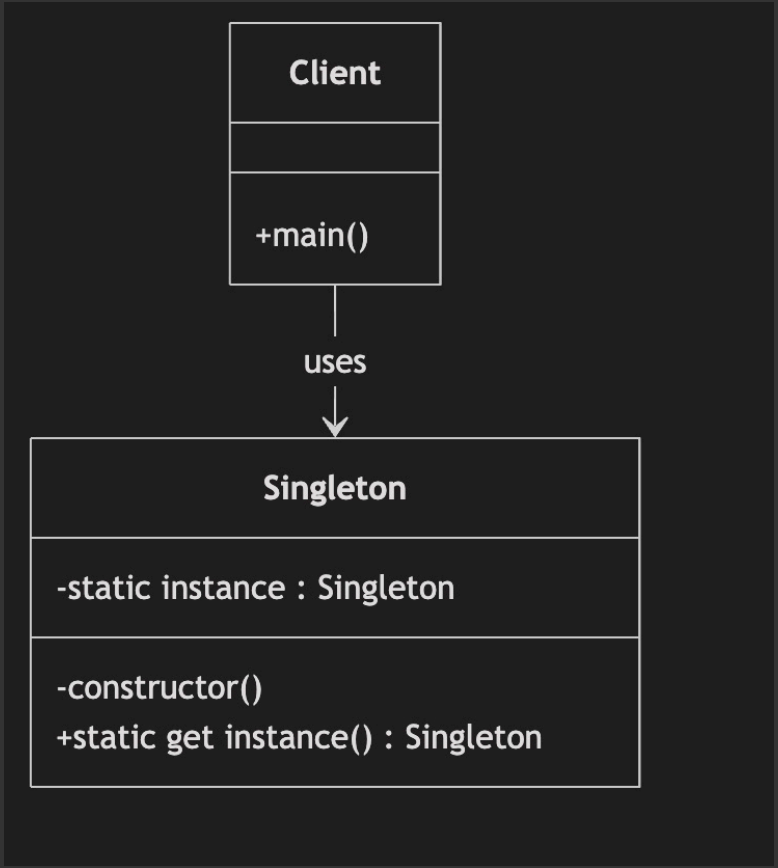
Objective:
Design a flexible beverage ordering system where:

Different types of beverages (e.g., lemon tea, green tea, coffee) can be created with their base cost.

The base beverages can be decorated with additional ingredients (e.g., honey, sugar, whipped cream), which will modify their cost dynamically.

The system should allow users to retrieve the final cost of a beverage after all customisations.

The design should be flexible enough to add more beverages or decorators in the future without modifying the core system.


Client
+main()
uses
Singleton
-static instance : Singleton
-constructor()
+static get instance() : Singleton

static ⇒ (a) class
or
(b) individual objects

new

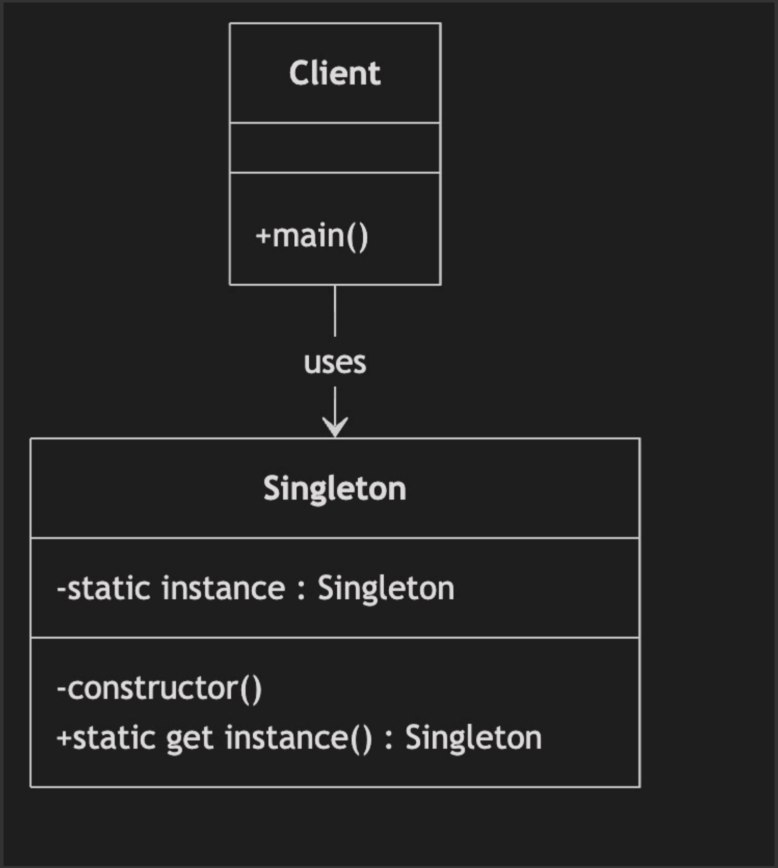
```
class myCalcul {
    static count = 0;
    static multiply(a, b) {

    }

}
```

static

→ object creation not req.

→ belongs to class, ! objects.

Singleton Design Pattern


Client
+main()
uses
Singleton
-static instance : Singleton
-constructor()
+static get instance() : Singleton

Problem Statement: Implement a Singleton Logger in TypeScript

Objective:

Create a logging utility using the Singleton design pattern to ensure that only one instance of the logger exists throughout the application. The logger should provide methods for logging different levels of messages: standard logs, warnings, and errors.

Requirements:

1 Implement a Logger class with a private constructor to prevent direct instantiation.

2 Provide a static method getInstance() that returns the single instance of the Logger.

3 Implement the following instance methods:

- log(message: string) – prints a standard log message.
- warn(message: string) – prints a warning message.
- error(message: string) – prints an error message.

4 Demonstrate that multiple calls to getInstance() return the same logger instance.

Problem Statement: Implement a Singleton Configuration Manager in TypeScript

Objective:
Create a Singleton Configuration Manager class in TypeScript that provides centralized access to application configuration. The configuration should be loaded only once and shared across the application.

Requirements:

Singleton Pattern

- Implement a ConfigManager class with a private constructor.
- Provide a static getInstance() method that returns the single instance of the class.

Configuration Storage

- The manager should store configuration as an object with the following structure:

```
    type Config = {
    apiUrl: string;
    port: string;
}
```

Access Configuration

- Provide a method getConfig() to return the entire configuration object.

Optional: Key-Based Access

- Provide methods get(key) and set(key, value) for individual key access.
- Optional requirement: restrict key to only the keys defined in the Config type (apiUrl or port) using keyof Config.

```
    get(key: keyof Config): string; // optional restriction
set(key: keyof Config, value: string): void; // optional restriction
```

Usage Example

```
const config1 = ConfigManager.getInstance();
console.log(config1.getConfig());

console.log(config1.get("apiUrl"));
console.log(config1.get("port"));

config1.set("apiUrl", "http://localhost:4000");
console.log(config1.get("apiUrl"));
```

Expected Behavior:

Only one instance of ConfigManager should exist.
Configuration should be loaded only once, regardless of how many times getInstance() is called.
get(key) and set(key, value) should optionally allow type-safe access to known configuration keys.

TV

mobiles

Laptop

Watches

Factory Design Pattern

↳ Produce → objects.

Problem Statement: Product Creation Using Factory Pattern

You are tasked with designing a scalable product creation system for an e-commerce platform.
The platform sells various types of products — such as Laptops and Mobiles — each having unique specifications.
As the platform grows, more product types (like TVs, Headphones, or Smartwatches) may be added in the future.
To avoid writing repetitive new statements and complex object initialization logic in multiple places, you must design a centralized factory class that is responsible for creating different product objects based on type.

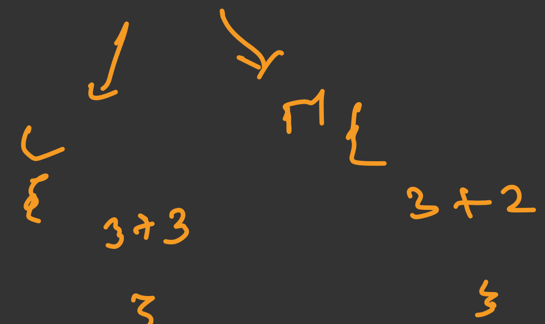
Requirements

1 Create a common Product interface with: <<Product>>

- name: string
- price: number
- getDescription(): string

2 Implement two product classes:

- Laptop with additional properties ssd, ram, and gpu
- Mobile with additional properties camera and battery



3 Create a ProductFactory class that:

- Exposes a static method createProduct(type, data) → Factory function.
- Returns the correct product object (Laptop or Mobile) depending on the type provided
- Throws an error for invalid product types

4 Demonstrate usage by:

- Creating a Laptop and a Mobile using the factory
- Displaying their descriptions using getDescription()

⇒ Factory Function

```
create Product( type, data ): Product {




}
```

has - a

Aggregation

Compos^{n}

Assoc^{n}

(uses)