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?

↓

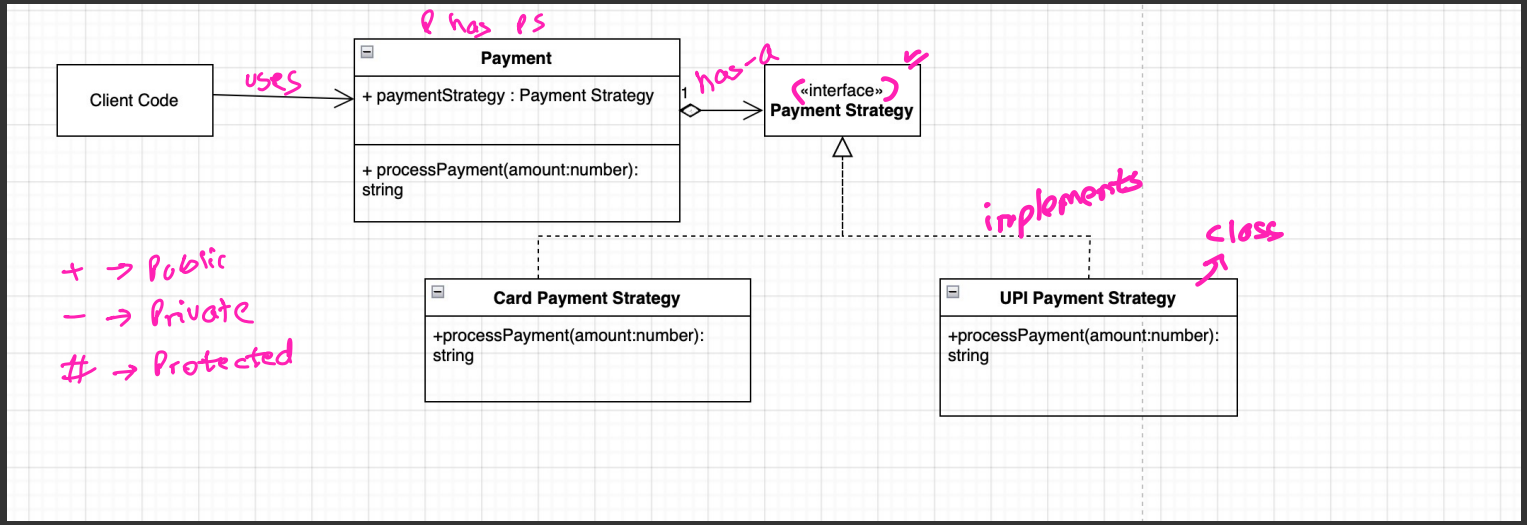
Basic (Fundamentals)

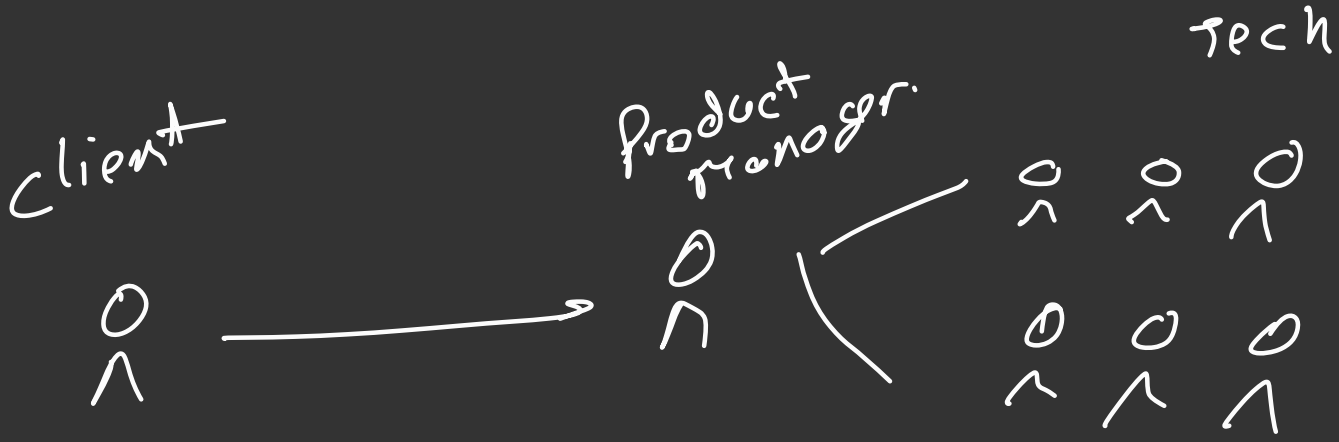
↓

OOPS + SOLID

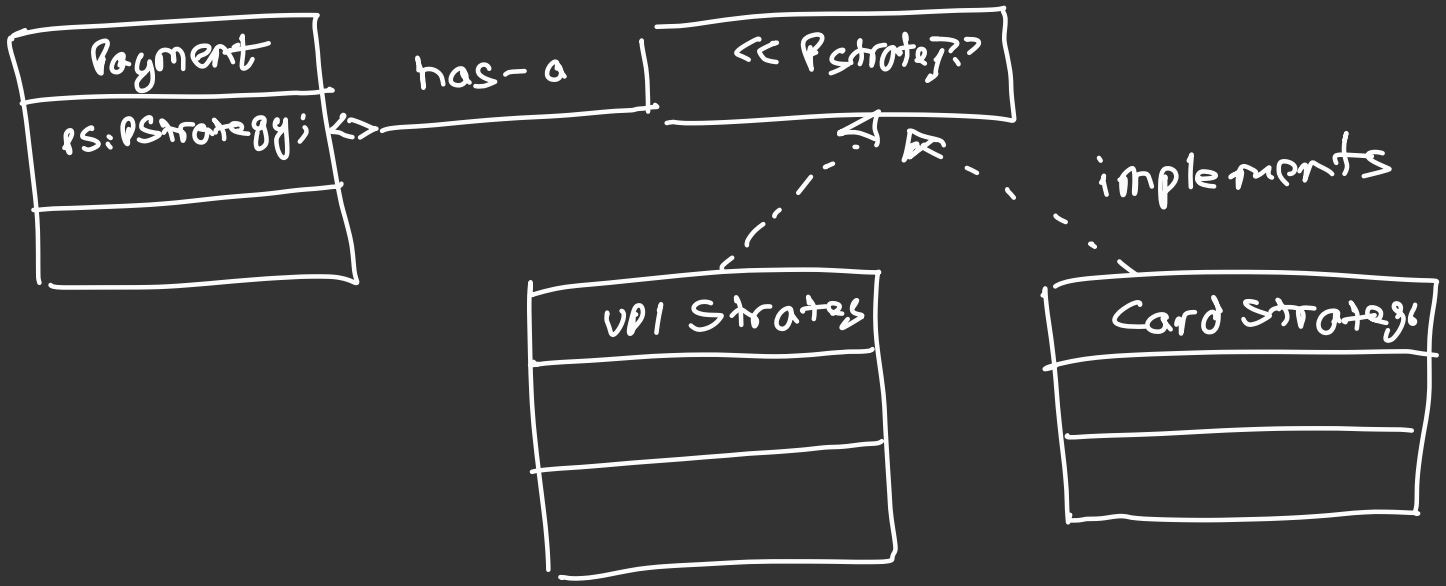
Strategy Design Pattern

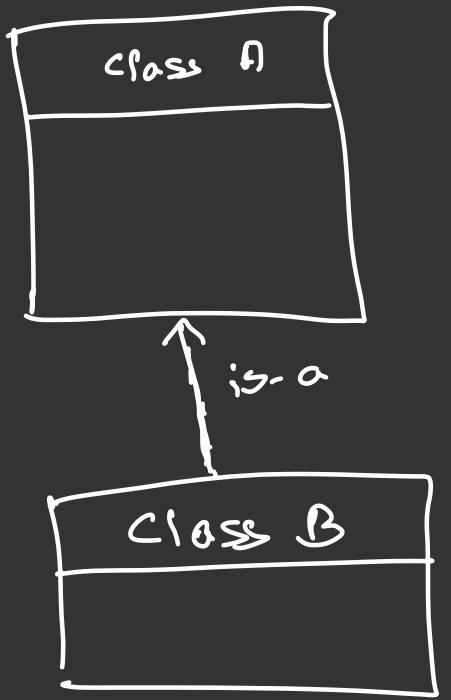
- Pre-defined Algorithms: Sorting, Discounting Logic, etc.
- Swithc between algorithms
- Reduce Code Duplication
- If 2-3 strategies, might be an overkill
- OCP
- Too many classes

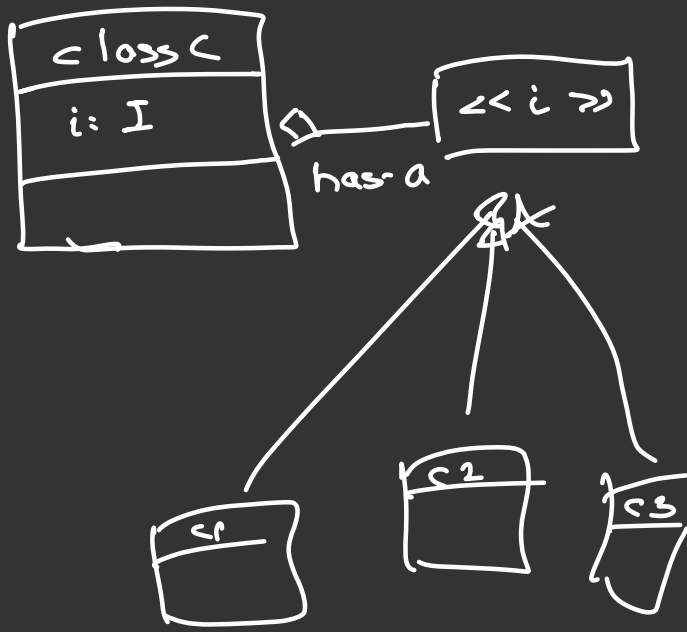

P has PS
Client Code
uses
Payment
+ paymentStrategy : Payment Strategy
+ processPayment(amount:number): string
1
has-a
«interface»
Payment Strategy
implements
class
Card Payment Strategy
+processPayment(amount:number): string
UPI Payment Strategy
+processPayment(amount:number): string
+ → Public
− → Private
→ Protected


Tech
Client
Product manager

Class Diagram of Strategy Design Pattern







website: draw.io → UML

=> Has-a.

Aggregation and Composition

weak

strong.

class A {

c: C → injected from outside.

}

class B {

c: C = new C(); → ownership. A

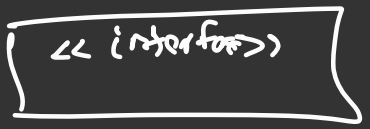
}

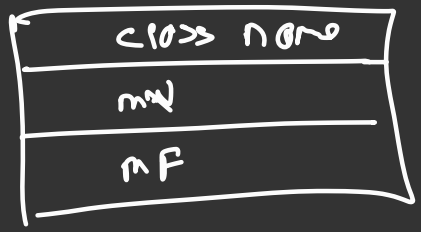
class C

}.

Class Diagram rules

- Inheritance
- Interface
- Aggregation (Hollow Diamond)
- Composition (Solid —)





\+ public
— Private
Protected

Compliant

Violⁿ

SOLID

LSP

↓

inheritance (is-a)

↓

true in L

NVI is-a.

Media

→ A

→ V

Audio output

→ A

→ V X

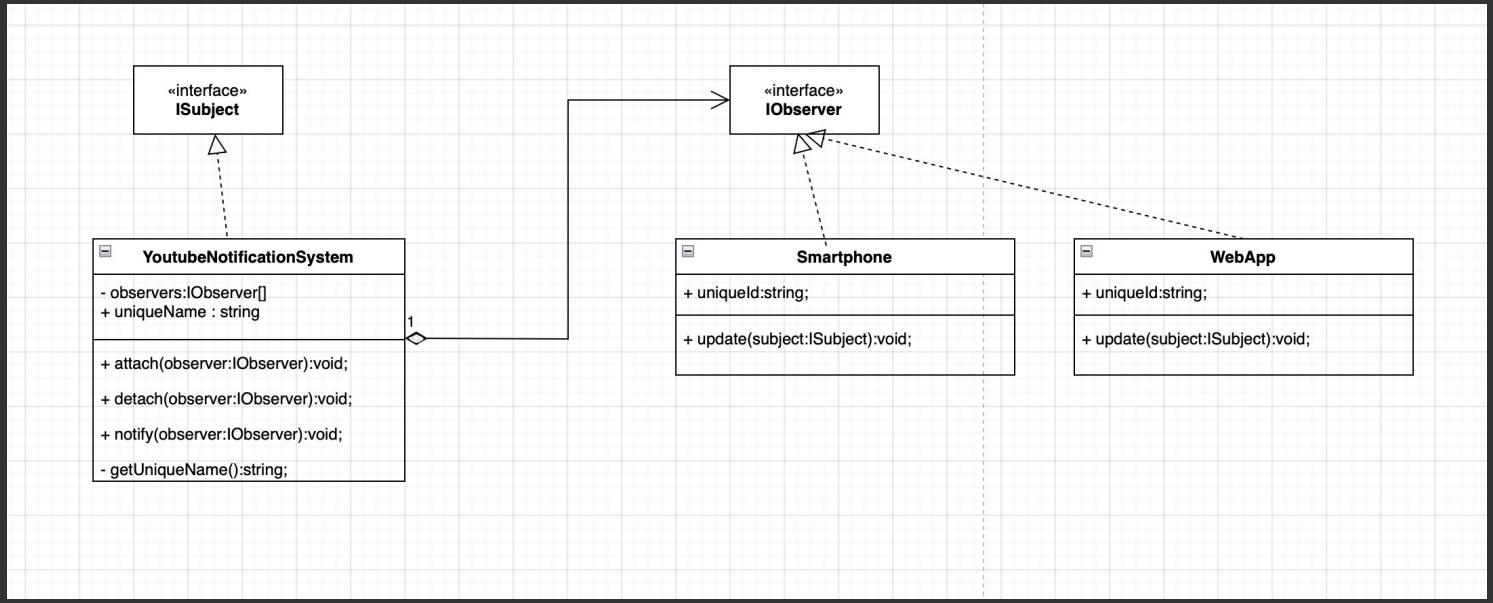
DIP

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;
1
Smartphone
+ uniqueId:string;
+ update(subject:ISubject):void;
WebApp
+ uniqueId:string;
+ update(subject:ISubject):void;