Lecture 40 : Wrapping Up S.D

Anti - Patterns

- Aisi cheeze joh hum galti se kor dete hai jiske wajah se baadmein problems aa skti hai

- First thing in it,

- **God Object**

↳ Aise objects jismein bahoot saare methods ho and humare application ka major chunk god object handle kar raha hai yaa uske through handle hoo raha hai.

Doubt : Is our orchestra class God object

→ So there are two scenarios to it -

1) Delegating : If class is just delegating the task to other class then it is not God object

2) Delegating with a twist : So our object is delegating the task but before delegating it does some work in respective methods

↳ It is a God object

Solution?

① Create multiple object and client has to call those and should know about all class

② Use orchestra class but with only delegation ⇒ a single entry point to easier task of client.

Ex: Api ~~ser~~ Microservice → Controller layer ⇒ waha saare

api's define hote hai

- **Spaghetti Code :**
  - Making object itna complex ki usmein naa koi entry point ho, no exit point.

  - Too much tightly coupled no clear paths, all objects depend on each other

  Problem Arise -
  ① Highly tight coupled
  ② Prone to error.

- **Hard-Coding things :**
  If we hard code the value like in
  ```
  m1(){
      string s = "Hello";
  }
  ```
  } Ab humne yeh maan liya ki s will always be hello.

  - What if it changes?

  - To avoid hard-coding things use constructor

- **Gold-plating / Over Engineering**
  - Trying to always acheive perfectionism in design
  - Handling case that will never arise
  - Applying patterns unnecessarily
  - Over complicated code

- **DRY - Do not ~~Reap~~ Repeat Yourself**

-Copy - Pasting same code
- Too much repetition
- Hard to maintain (changes needed in everyplace it is used)

How to Avoid?
- Extract repeated code into seperate logic/ method / class
- Use utility class

- **Constructor Overloading / Telescoping Constructor**
↳Creating too many constructors

solution ? => Builder Design Pattern

- **Overuse of Getters / setters**
- Providing getters / setters for all private members by default.

- Forgets the main purpose of private members

- should be used with validation / control

- Not every variable needs getter/setter.

- **Premature Optimization**

- Optimizing code too early

Principle: "Make it work, then make it fast"

- Ex: Brute Force solution First, then optimise for performance.

- **Overuse of Inheritance**

- Leads to complex hierarchies

- Causes tight coupling

- Alternatives
  - → Composition
  - → Strategy Pattern
  - → Visitor Pattern
  - → Bridge Pattern
  - → Inheritance — only for abstract class

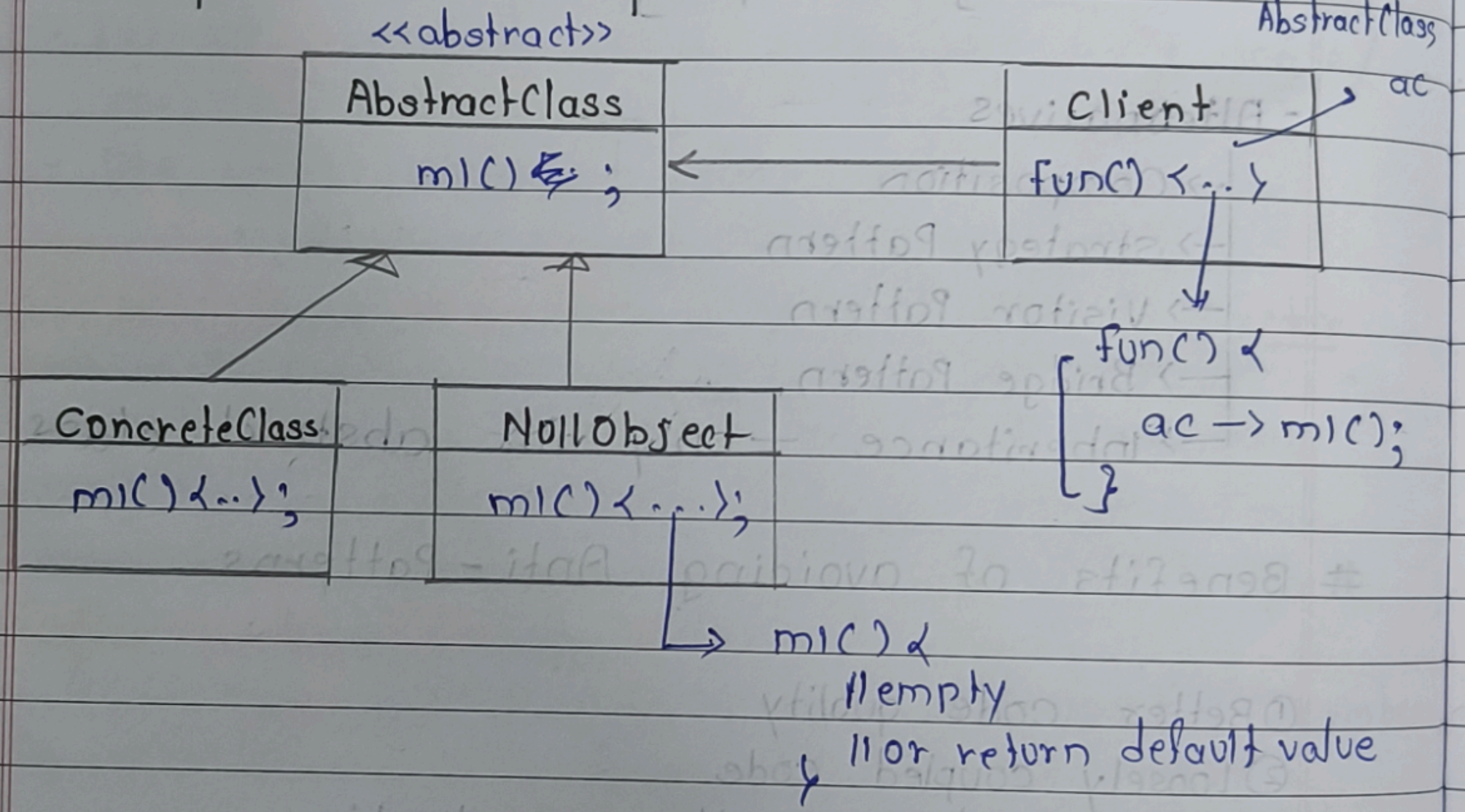
Benefits of avoiding Anti-Patterns

1. Better code quality
2. Loosely coupled code
3. Improved design pattern application
4. Easier new feature integration

Null Object Pattern

- Purpose
  - Avoid null checks [if (obj == null)]
  - Prevents NullPointerException
  - Replaces conditionals with polymorphism

- Concept with Example



- Create a 'Null Object' class
- Inherit from our abstract class
- method in null object does nothing or return default values (like zero, empty string)
- Client always receives an object reference (concrete / null object)
- Avoids Liskov subscription Principle break

* Why Null Object ?

- null - Reference points to nothing in heap ∴ throws error at runtime

- Null Object - Reference points to an empty object in heap

Example where we used our Null Pattern

1. ~~Object~~ Strategy Design Pattern
   we learnt it through Robot example -

- In it we created class - NoFlyBehaviour

- Robot has IFlyStrategy Object

- If robot doesn't fly, we assign NoFlyBehaviour whose fly() method does nothing

2. Command Design Pattern
   Learnt through Remote Control Example

   ↳ we initialized button with NoCommand

   ↳ NoCommand class's execute() method does nothing.