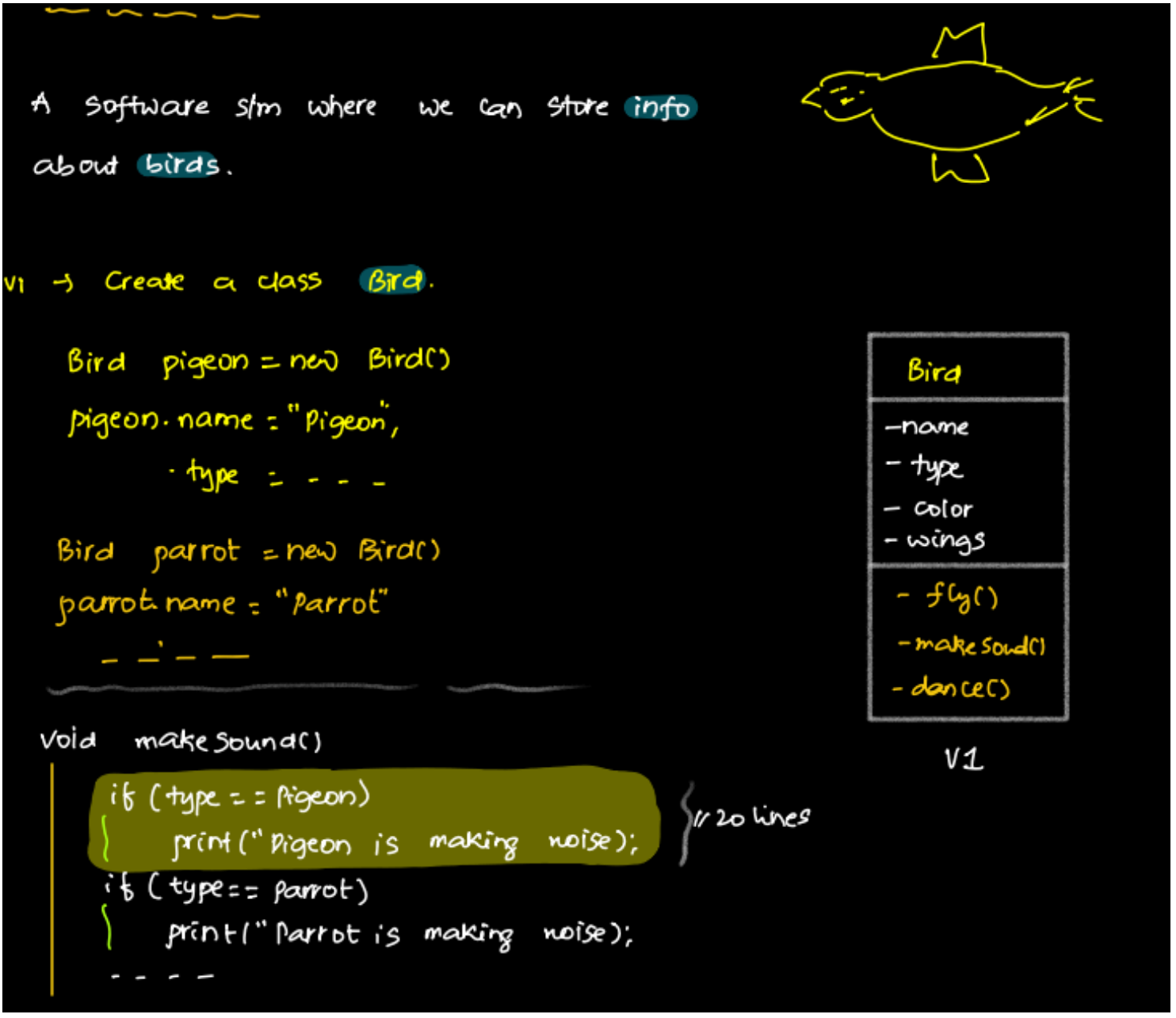
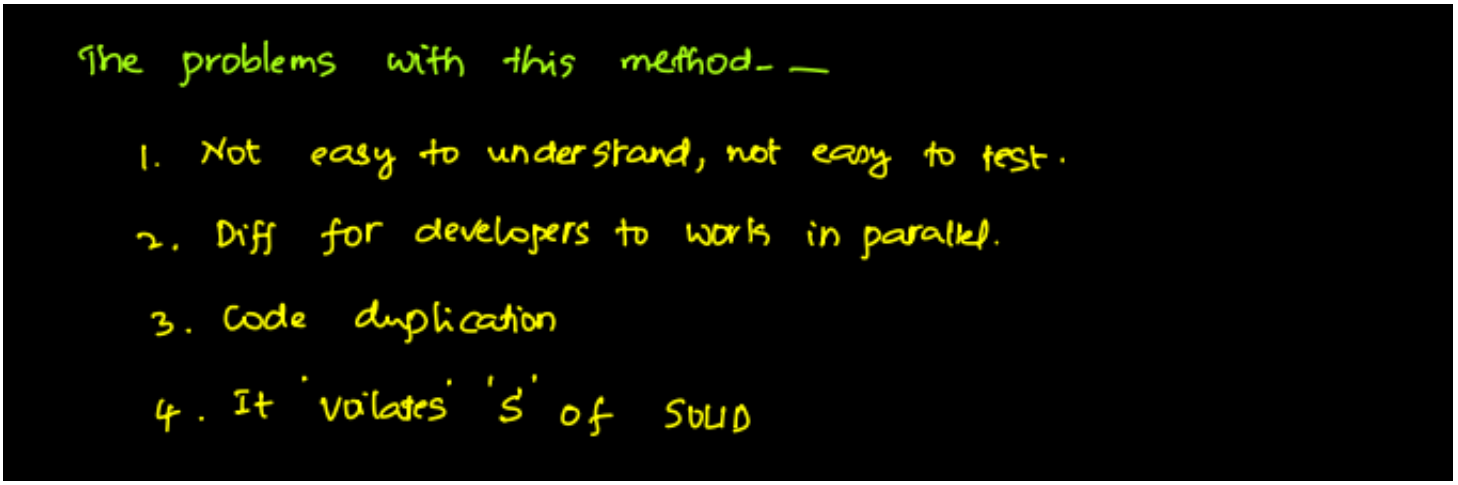
If you have any questions or suggestions regarding the notes, please feel free to reach out to me on WhatsApp: **Sanjay Yadav Phone: 8310206130**

Backend LLD-2: SOLID-1: SRP and OCP - Jan 09

- **Design a bird**
  - creating a simple bird class, will it do the job ?
  - The problems in one of the methods of bird class.
- **Solid Principles**
- **Single Responsibility Principle (SRP)**
  - How to identify violations of SRP
- **Open Closed Principle (OCP**
  - Making bird class abstract

Design a bird

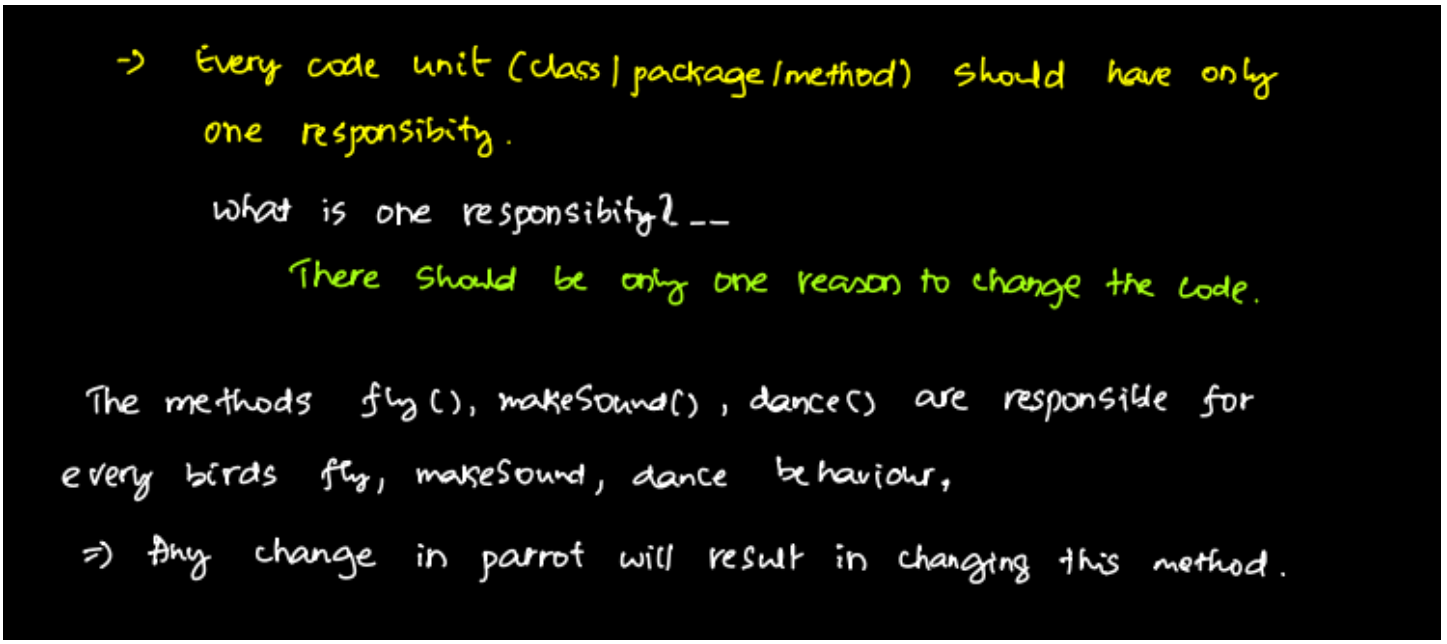




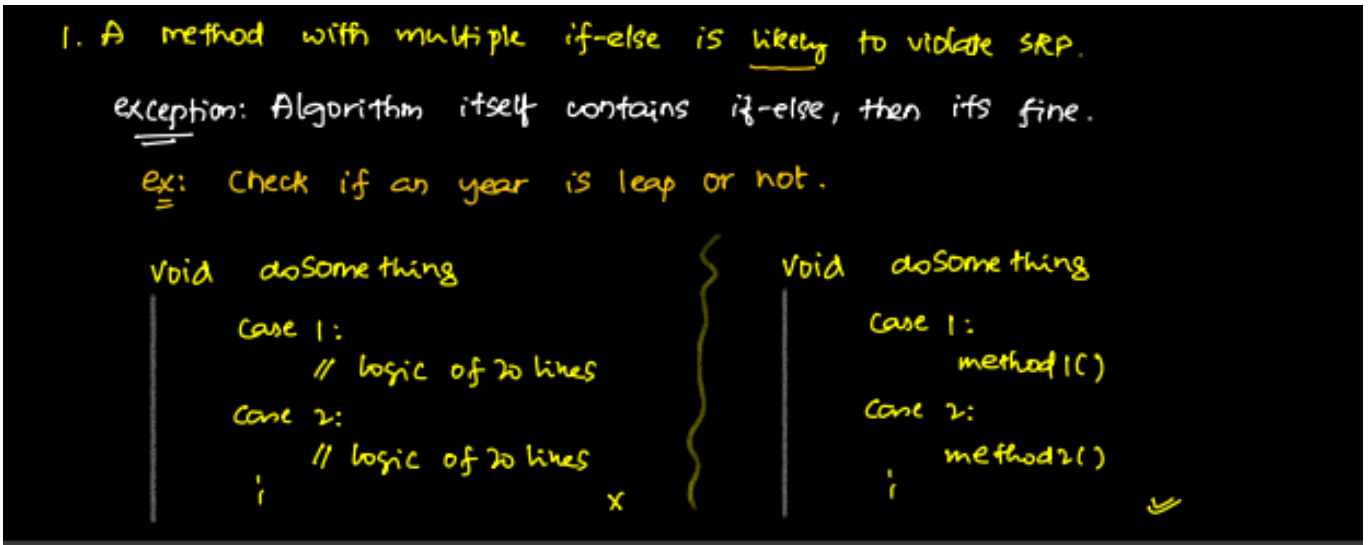
Solid Principles -> The SOLID principles are a set of design principles for writing maintainable and scalable software. These principles were introduced by Robert C. Martin and are widely used in object-oriented programming. The SOLID acronym stands for:

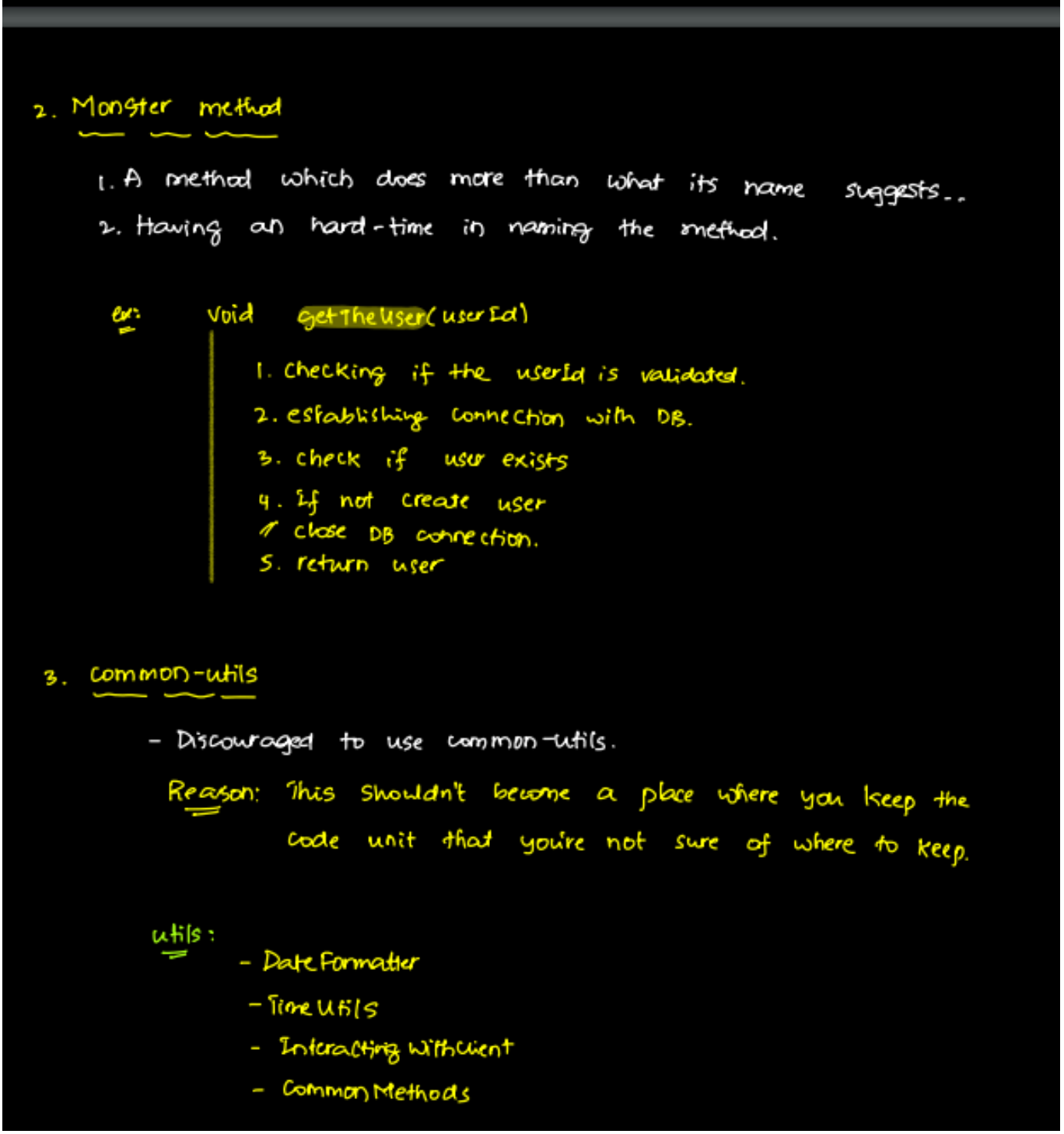
S -> Single Responsibility Principle (SRP)
O -> Open/Closed Principle (OCP)
L -> Liskov Substitution Principle (LSP)
I -> Interface Segregation Principle (ISP)
D -> Dependency Inversion Principle (DIP)

Single Responsibility Principle (SRP) -> A class should have only one reason to change, meaning that it should have only one responsibility or job. This principle encourages the idea that a class should focus on doing one thing well.

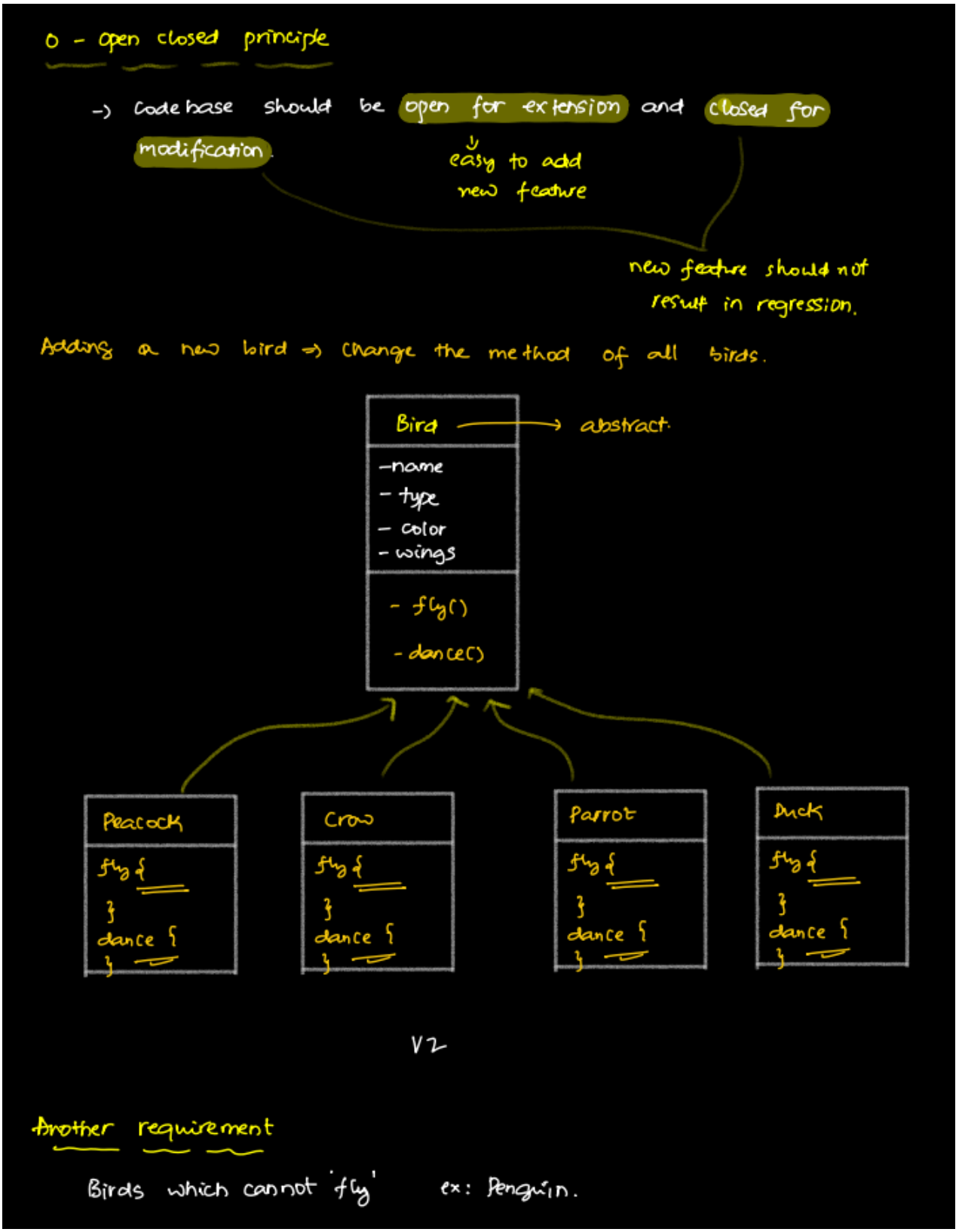


How to identify violations of SRP

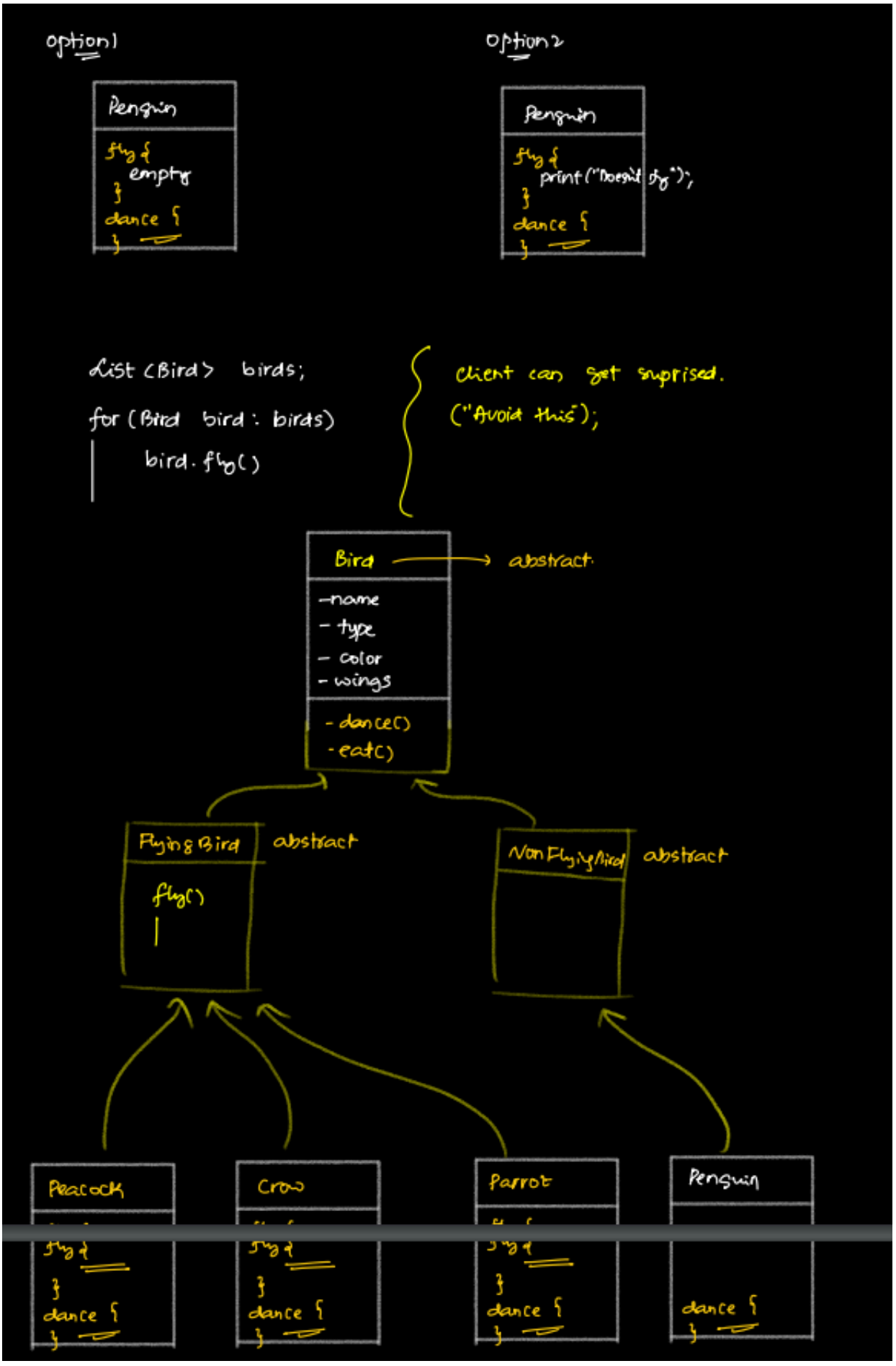




O -> Open/Closed Principle (OCP) -> Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. This means that you should be able to add new functionality without changing existing code.

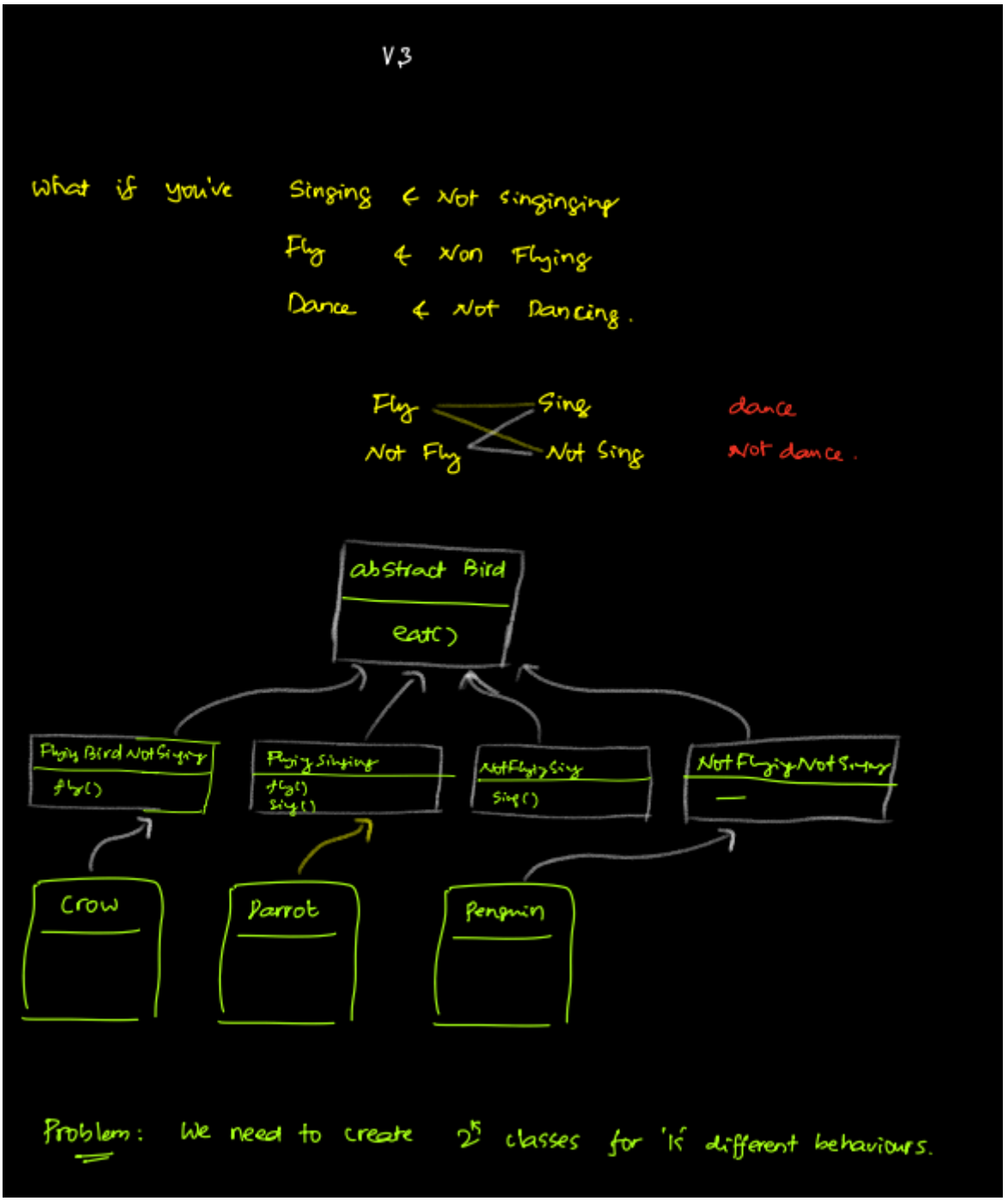


If we add the word "penguin" to the system, it might cause a problem because penguins can't fly. We'd have to create a way for penguins to fly, either leaving it empty or showing an error. However, clients using the system wouldn't know about flying birds, and if they expect 10 birds, they'd be surprised to find only 9 because penguins can't fly.



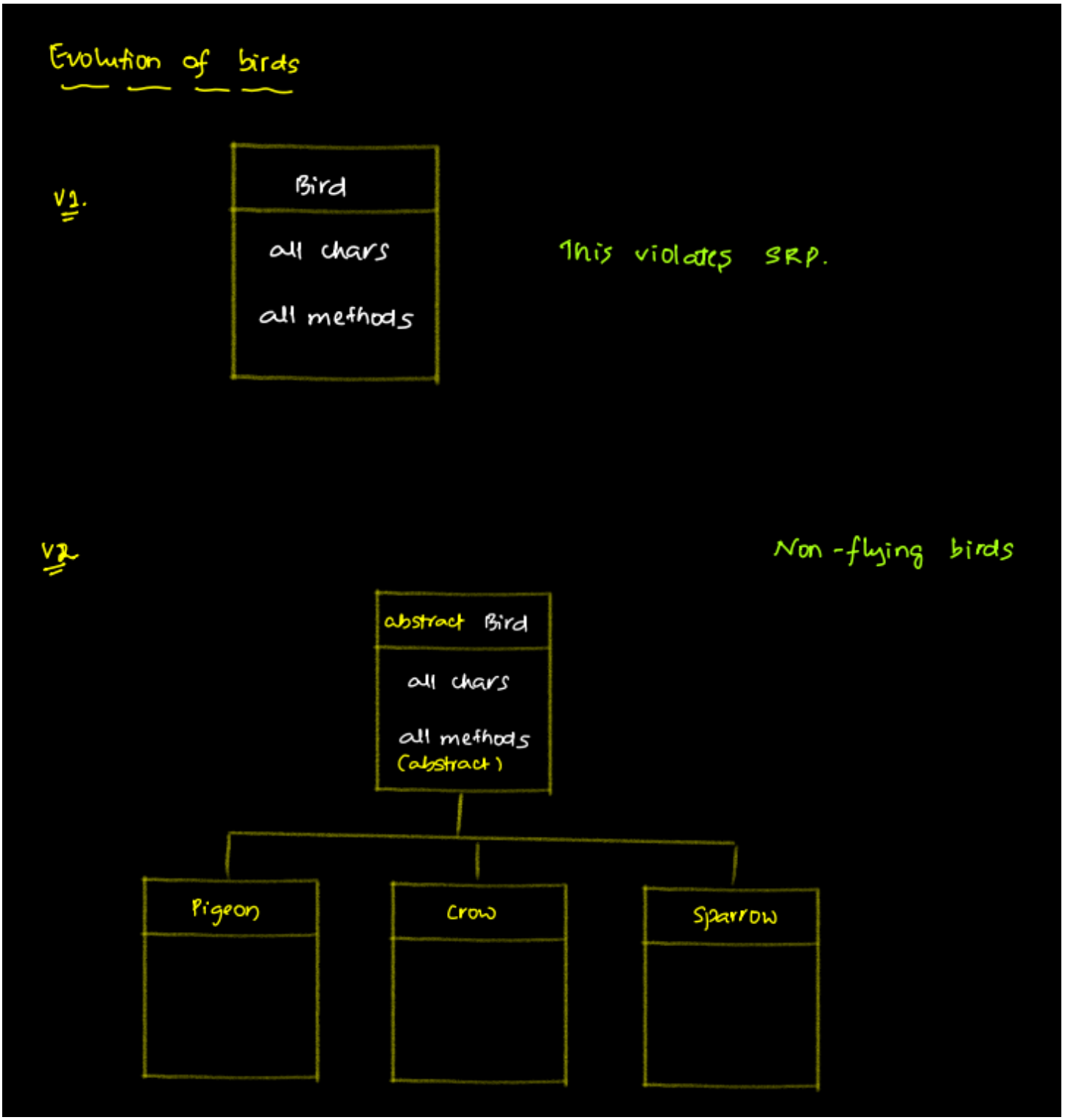
To meet the requirement of adding penguins, we'll introduce an additional layer using abstract classes. We'll split it into two parts: FlyingBird and NonFlyingBird, and then extend these abstract classes.

The solution mentioned earlier introduces another problem. Adding one more layer of abstraction using an abstract class requires multiple combinations, as illustrated below. This, once again, raises concerns regarding maintainability and code reusability.

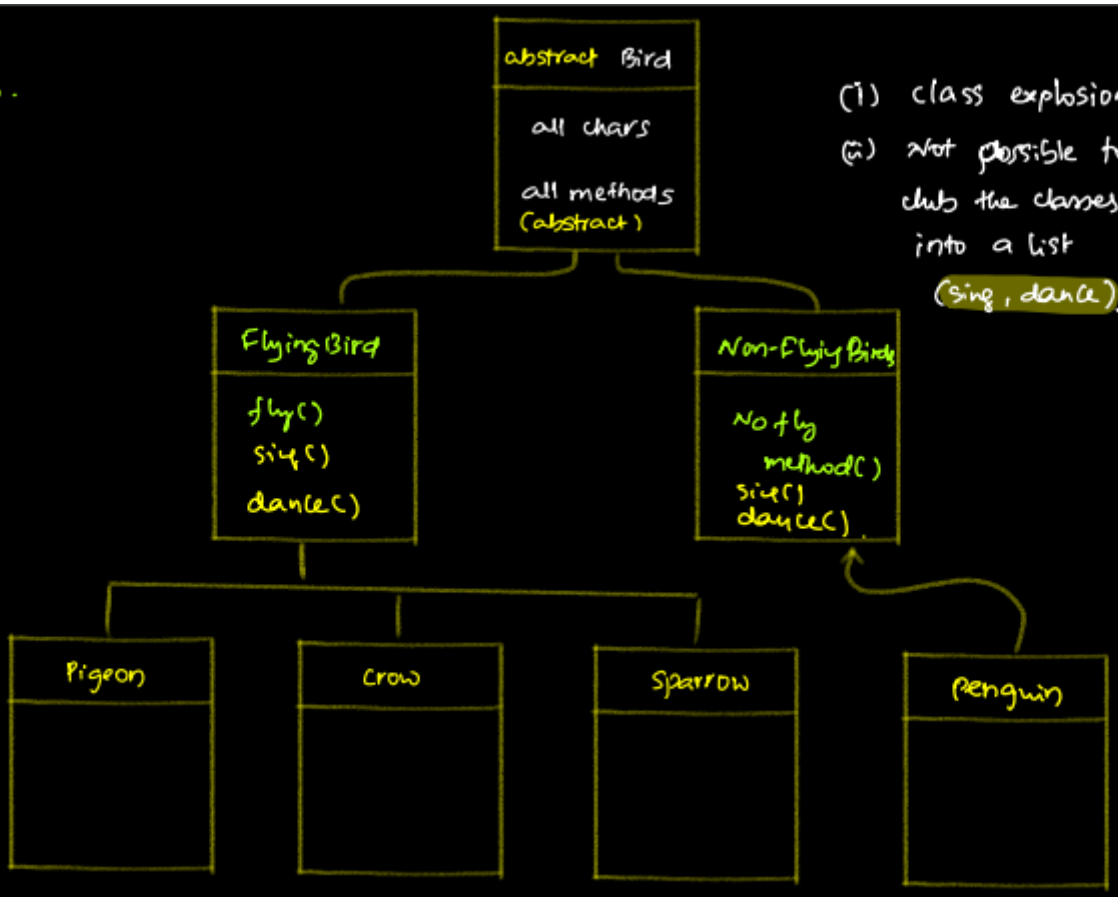


Backend LLD-2: Liskov's,this Interface Segregation, Dependency Inversion

- **Interfaces to solve class explosion problem**
- **Liskovs substitution principle**
- **Interface segregation Principle**
- **Code duplication problem, intro to Dependency Inversion**
- **Dependency inversion crisp definition**
- **Dependency Injection**



V3.



(i) class explosion

(ii) Not possible to club the classes into a list (sing, dance).

Problem Statement

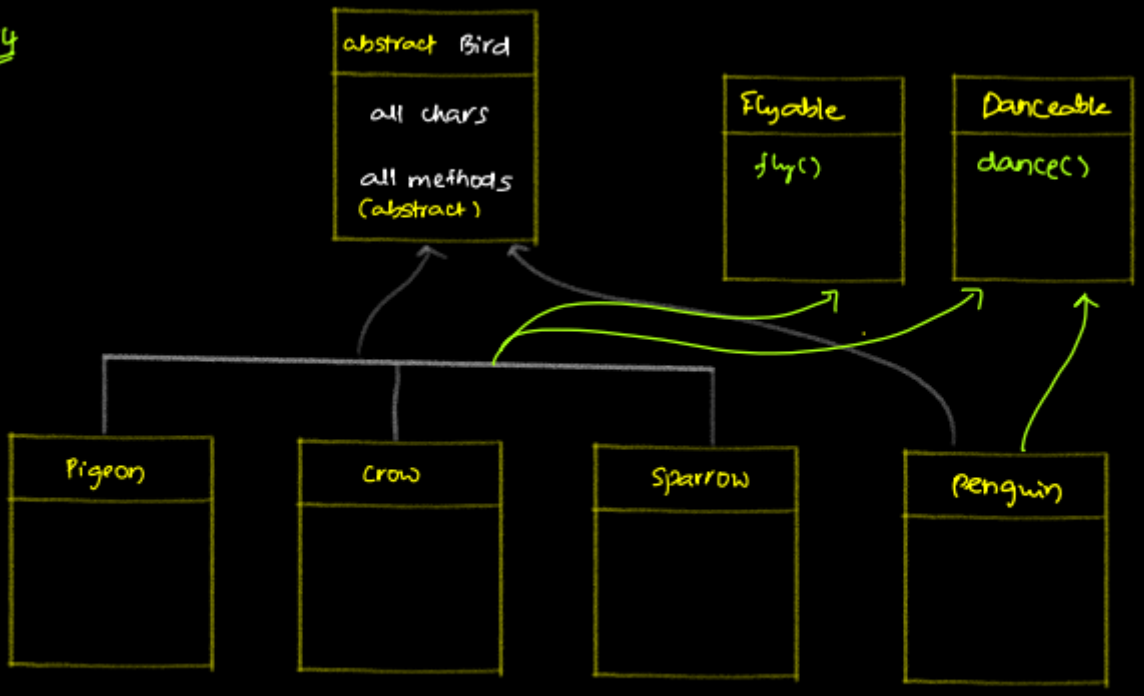
Some birds demonstrate a behaviour where some doesn't.

Expectations:

1. The birds which have that behaviour should have the method.
2. The class explosion should not happen.
   or I should be able to create a list of birds that have a particular behaviour.

Issue: We're trying to tie the blueprint of a behaviour to a class interface.

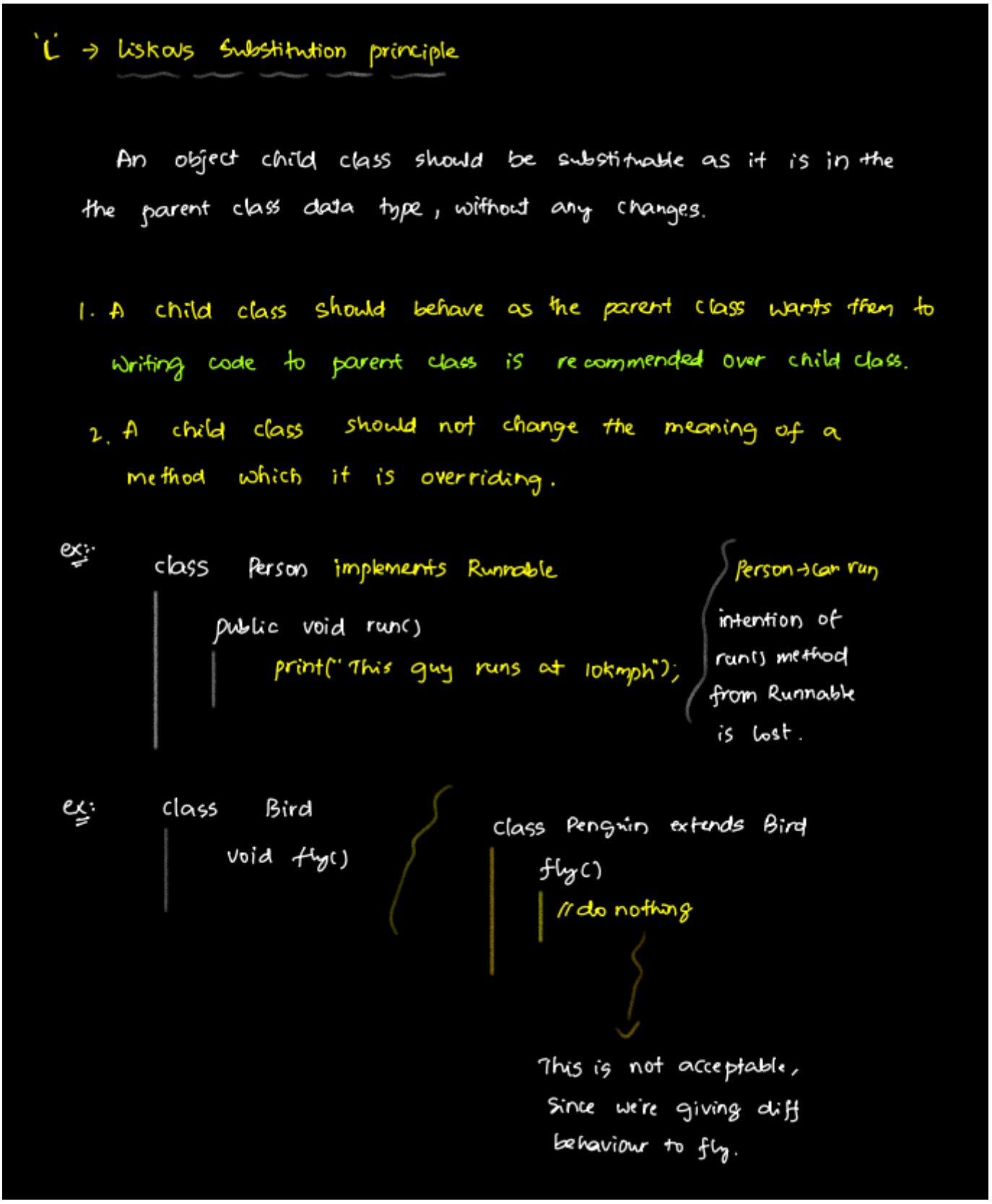
V4




```
class Pigeon extends Bird implements Flyable, Danceable
    fly() {

    }
    dance() {

    }
    eat() {

    }
```

I can create List<Flyable> allFlyingBirds;

Liskov Substitution Principle (LSP) -> Subtypes must be substitutable for their base types without altering the correctness of the program. In other words, objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program.



```
class Animal
    void eat()
        print(Animal is eating);
```

```
class Dog extends Animal
    Void eat()
        print("Dog is eating")
        print("Dog is sleeping")
```

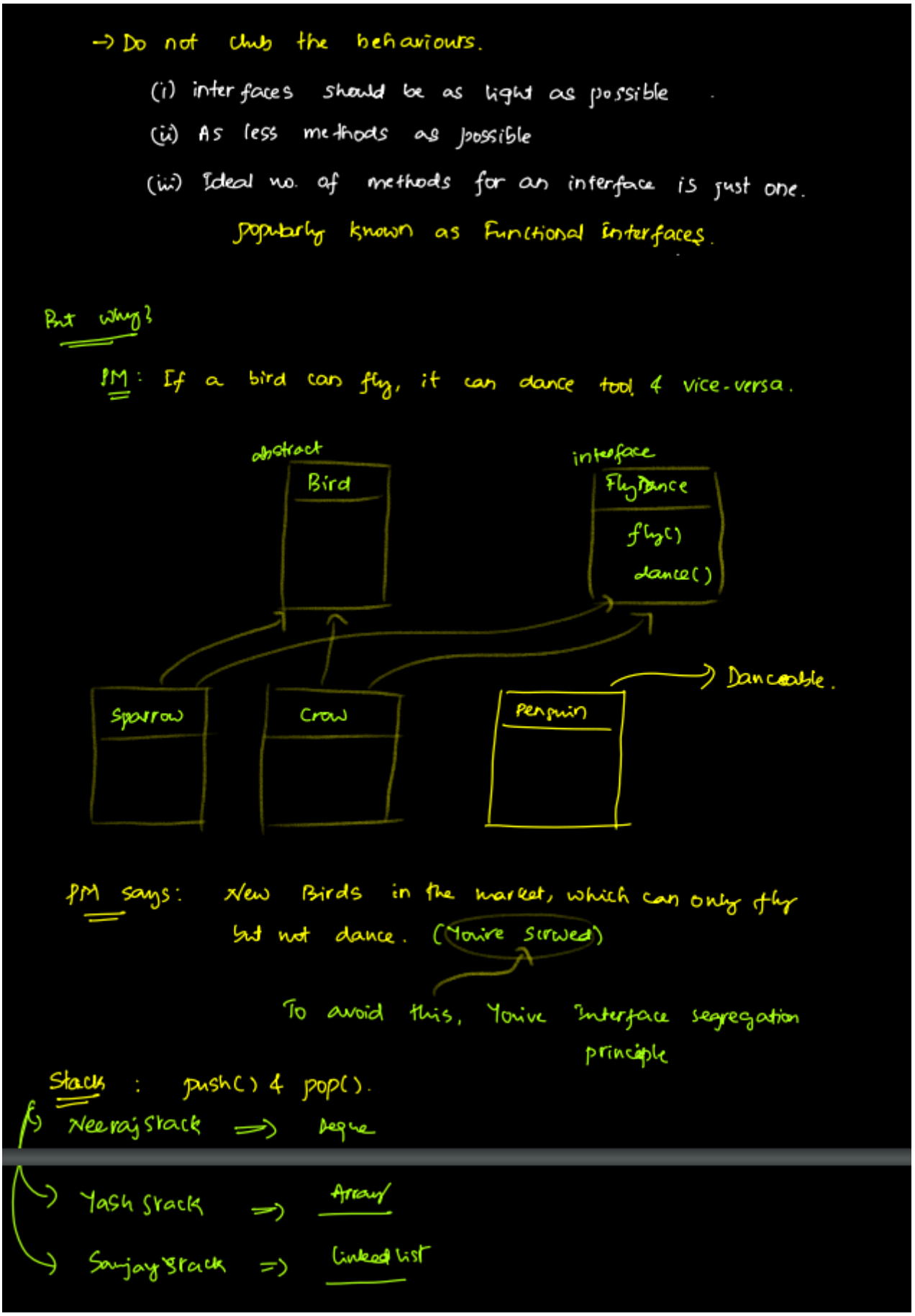
```
Main
    doSomething(Animal animal)
        animal.eat()
```

```
List<Integers> numbers = new ArrayList<>()
                         new LinkedList<>()

numbers.add(      )

numbers.remove(      )
```

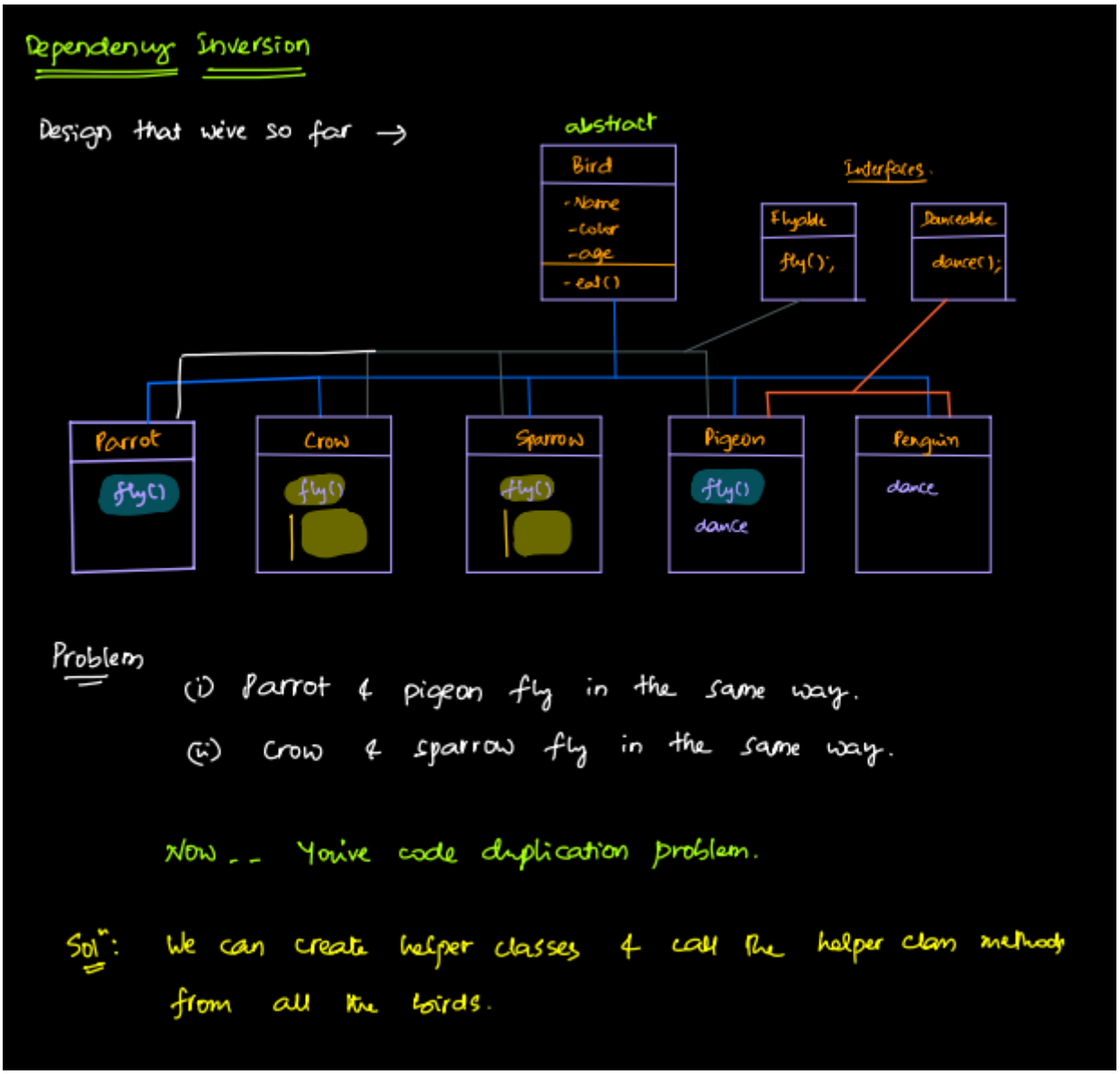
```
Animal
    getAnimal()
        // get animal from DB
```

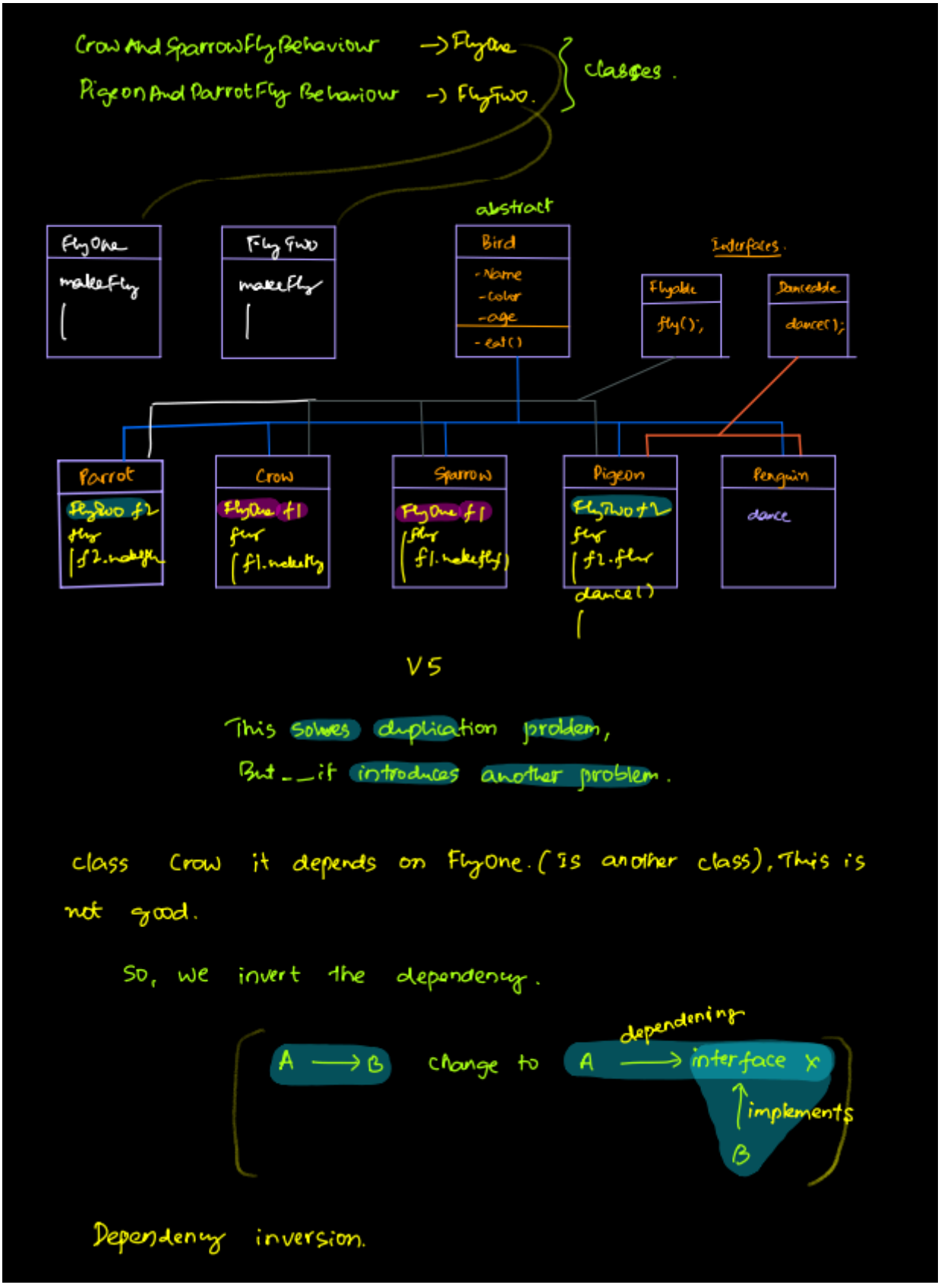
```
Dog
    getAnimal()
        // get dog from db -> doesn't exist.
        // create the dog
           & return it
```

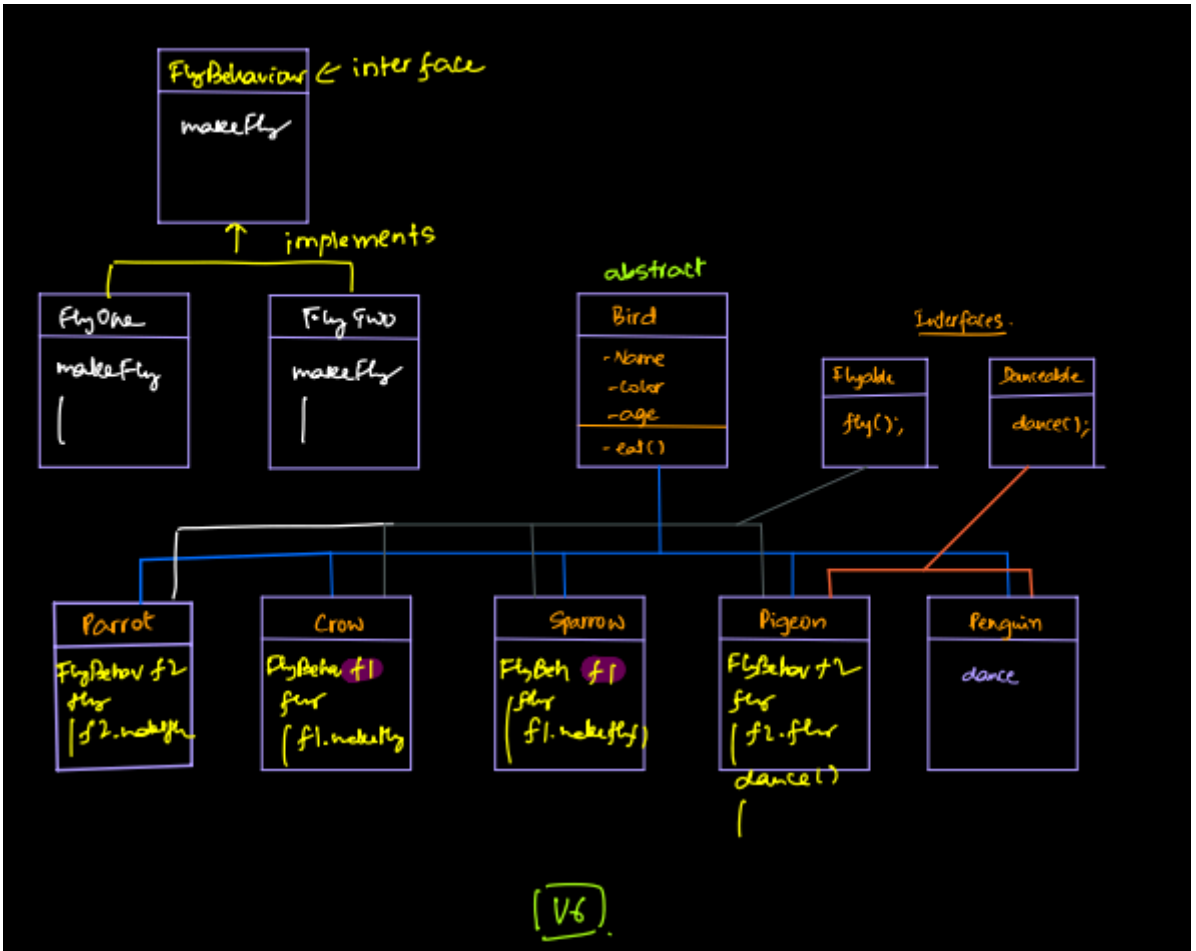
Interface Segregation Principle (ISP) -> A class should not be forced to implement interfaces it does not use. This principle encourages the creation of small, specific interfaces rather than large, general-purpose ones.



Dependency Inversion Principle (DIP) -> High-level modules should not depend on low-level modules. Both should depend on abstractions. This principle promotes the use of abstractions (interfaces or abstract classes) to decouple high-level and low-level modules, making the system more flexible and easier to maintain.



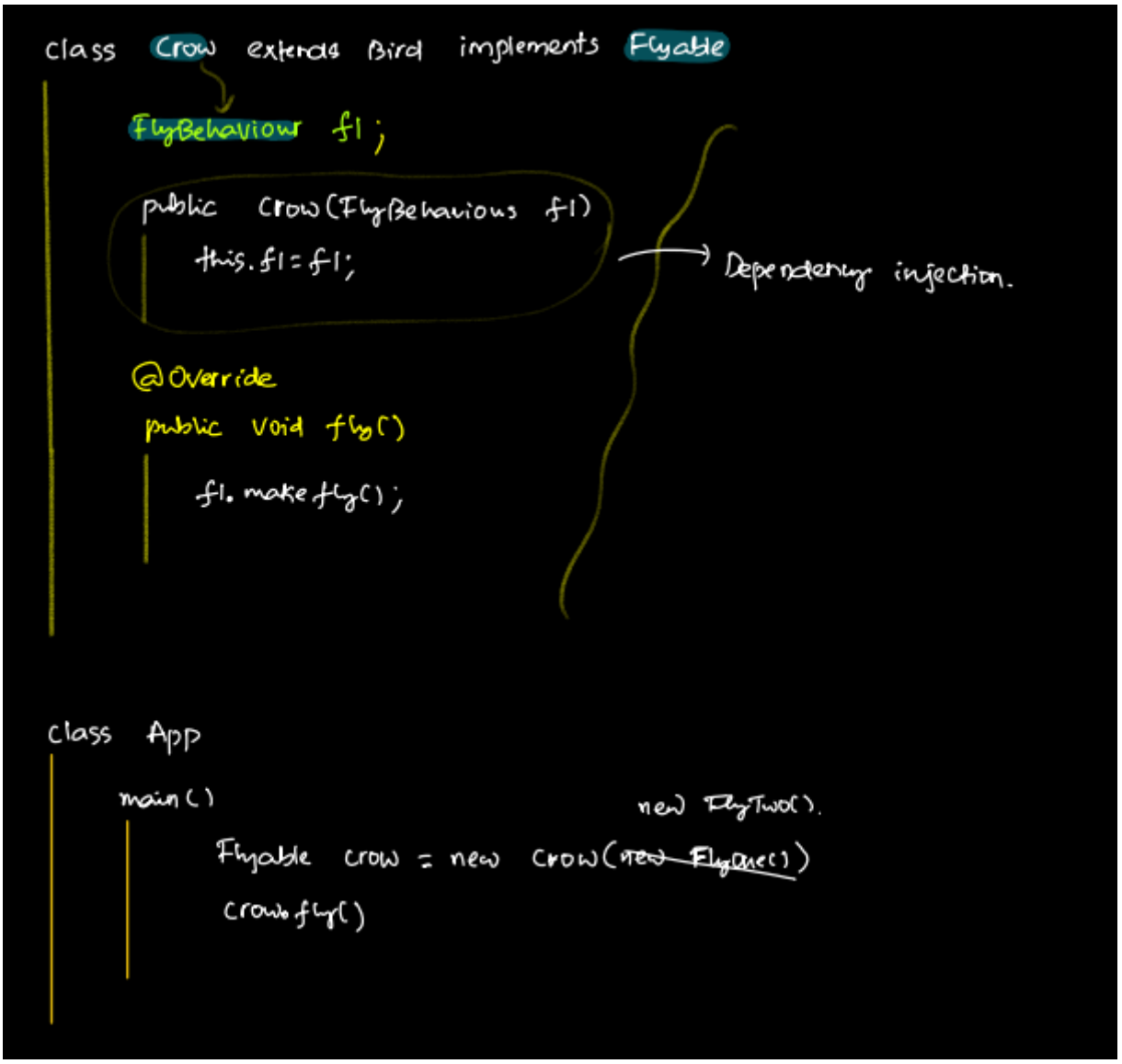

Crow And SparrowFlyBehaviour -> FlyOne
Pigeon And ParrotFly Behaviour -> FlyTwo.
classes.
abstract
Bird
- Name
- Color
- age
- eat()
Interfaces.
Flyable
fly();
Danceable
dance();
FlyOne
makeFly
FlyTwo
makeFly
Parrot
Crow
Sparrow
Pigeon
Penguin
dance
dance()
V5
This solves duplication problem,
But__it introduces another problem.
class Crow it depends on FlyOne.(Is another class), This is not good.
So, we invert the dependency.
A -> B
change to
A -> interface X
dependening
implements
B
Dependency inversion.



V6.

```
class Crow extends Bird implements Flyable

    FlyBehaviour fl = new FlyOne()

    @Override
    public void fly()
        fl.makeFly();
```

Problem: Crow now flies in FlyTwo way, not in flyOne() way anymore.


class Crow extends Bird implements Flyable
FlyBehaviour fl;
public Crow(FlyBehaviour fl)
this.fl = fl;
Dependency injection.
@Override
public void fly()
fl.makeFly();
class App
main()
new FlyTwo().
Flyable crow = new Crow(new FlyOne())
crow.fly()

Backend LLD & Projects 2: Design Patterns: Introduction and Singleton - Jan 13
https://github.com/kkumarsg/lld/blob/main/src/main/java/designpatterns/singleton/App.java

- **What are design patterns ?**
- **Types of design patterns**
  - **Creational**
  - **Structural**
  - **Behavioural**
- **Singleton design pattern**
- **Why singleton is needed ?**
- **How to create single class**
- **How to create single class**
  - **Observation 1 : Problem lies in constructor being exposed to everyone**
  - **Observation 2 : Making the constructor will not allow anyone to create an instance :(**
  - **Observation 3 : Expose another public method via which we can access the private constructor ? But, wait.. Make that method 'static'**
- **Final solution for non threading application.**
- **What happens if the application is multithreaded —**
  - **solution 1 for multithreaded application : Making the whole method static -> will result in slow performance. (LAZY Initialisation)**
  - **solution 2 for multithreaded application : Initialise the instance field at the time of declaration itself -> will result in increasing start time of application. (EAGER Initialisation)**
  - **solution 3 for multithreaded application: Double check locking, This is expected for interviews**

What are design patterns -> Design patterns are general reusable solutions to common problems that occur in software design. They represent best practices evolved over time by experienced software developers. These patterns can speed up the development process by providing tested, proven development paradigms.

Design patterns aren't blueprints or templates, but rather guidelines for solving certain types of problems. They facilitate communication between designers by providing a common vocabulary and can be a helpful tool for creating maintainable and scalable software systems.

What are design pattern?

- Software systems
- Something that occurs frequently.

Well established solutions to commonly occuring usecases in Software system design.

They help you to implement solid principles.

Types of design patterns [LLD].

1. Creational
   - → How to create an object
   - → How many objects to create.

2. Structural
   - → How the class should be structured..
   - → Attributes of a class
   - → How can one class interact with other classes.

3. Behavioural
   - → Methods
   - → How to create/code methods.

Creational Patterns:

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

Creational design patterns

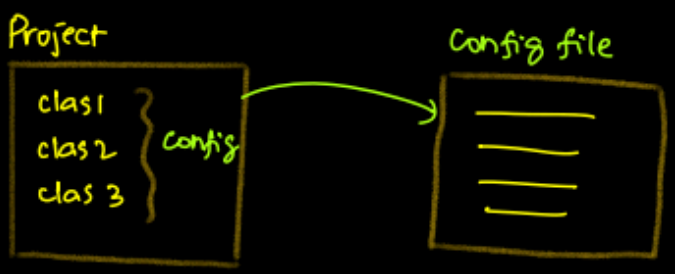
1. Singleton 2. Builder 3. Factory 4. Prototype.

Singleton [extremely important q^n in interviews].

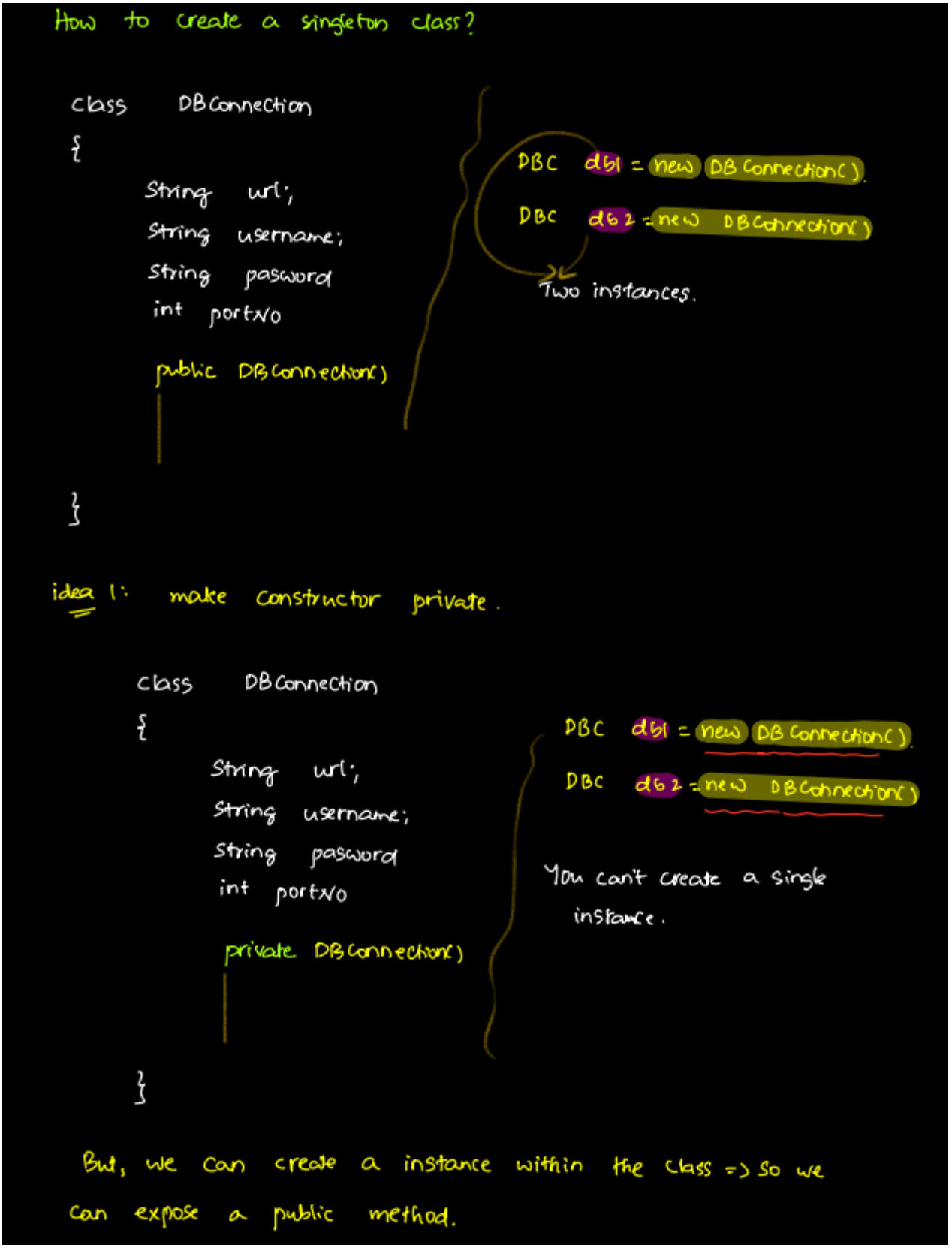
If for a particular class we can only create single instance of it. => Then it's a singleton class...

Why singleton?

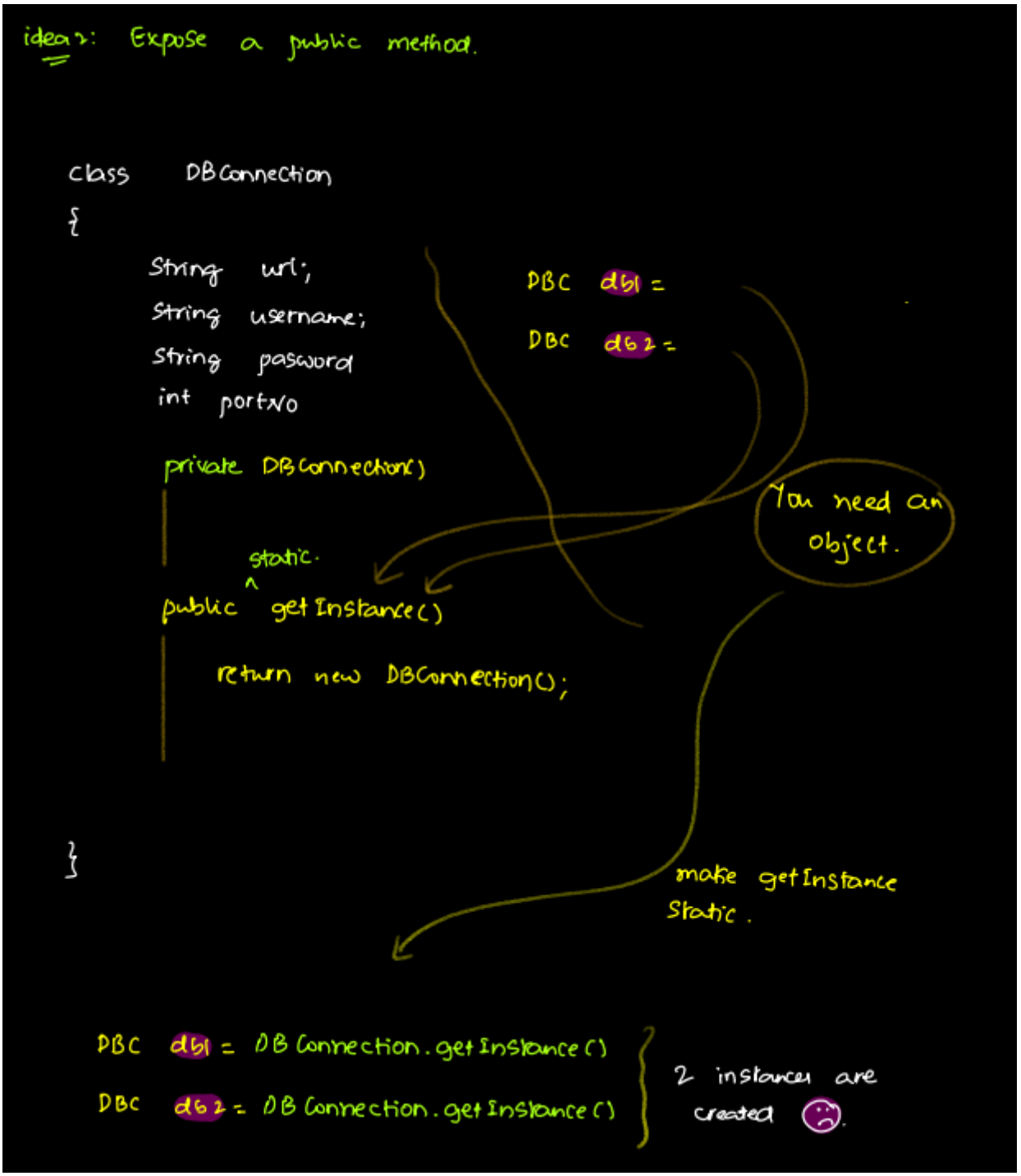
Sometimes only one instance is enough.



Interacting with DB,



As seen in the screen above, if we can create multiple instances using the constructor, it means it is not a singleton. Attempting to make the constructor private will also not work because we are unable to create any instances.



As observed in the screen above, if we make the constructor private and expose a getInstance method that returns an object, it becomes necessary to create an object to access this method. However, since the instance is private, we cannot do this. To address this, we make the getInstance method static. Now we can access this method using the class, but there is still the possibility of creating multiple instances.

To solve the problem, we set a condition like below that checks if an instance already exists. If it does, we return the existing instance; otherwise, we create a new one and assign it to instance. This works fine when there's only one thread of execution. However, in situations with multiple threads, there's a risk that two threads might simultaneously check the condition, leading to the creation of separate two instances.

SOLUTION

```
class DBConnection
{
    private static DBConnection dbC = null;

    String url;
    String username;
    String password
    int portNo

    private DBConnection()

    public static getInstance()
        if (dbC == null)
            dbC = new DBConnection();
        return dbC

}
```

#123.

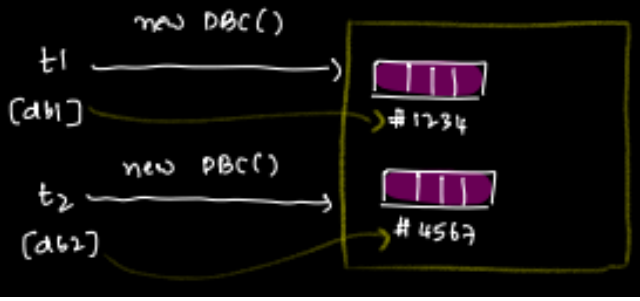
```
DBConnection db1 =
    DBConnection.getInstance();
DBConnection db2 =
    DBConnection.getInstance()
```

dbC = #123 [some address]

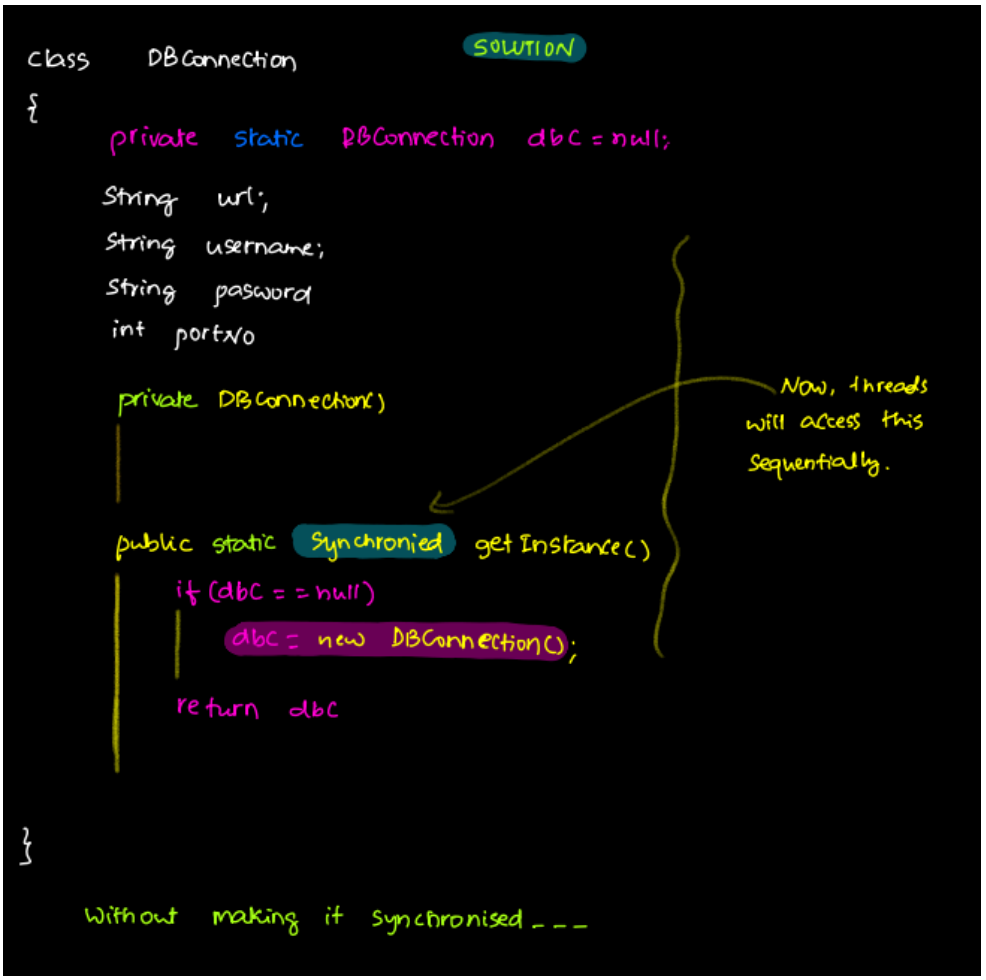
Works great for single threaded app^.

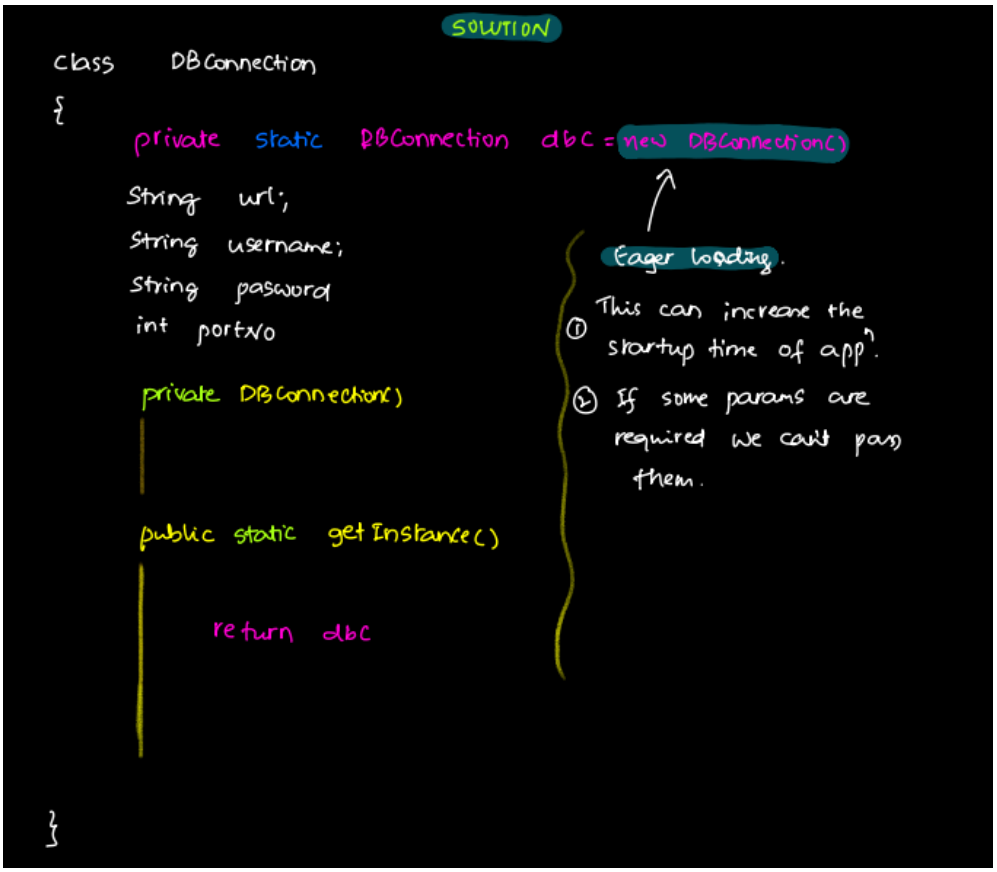
What about multi-threaded app^? ⇒ Doesn't work.

```
db1 = DBConnection.getInst()        t1
    if (dbC == null)
        dbC = new DBConnection();
    return dbC
    dbC = #1234

db2 = DBCon.getInst                 t2
    if (dbC == null)
        dbC = new DBConnection();
    return dbC
    dbC = #4567
```

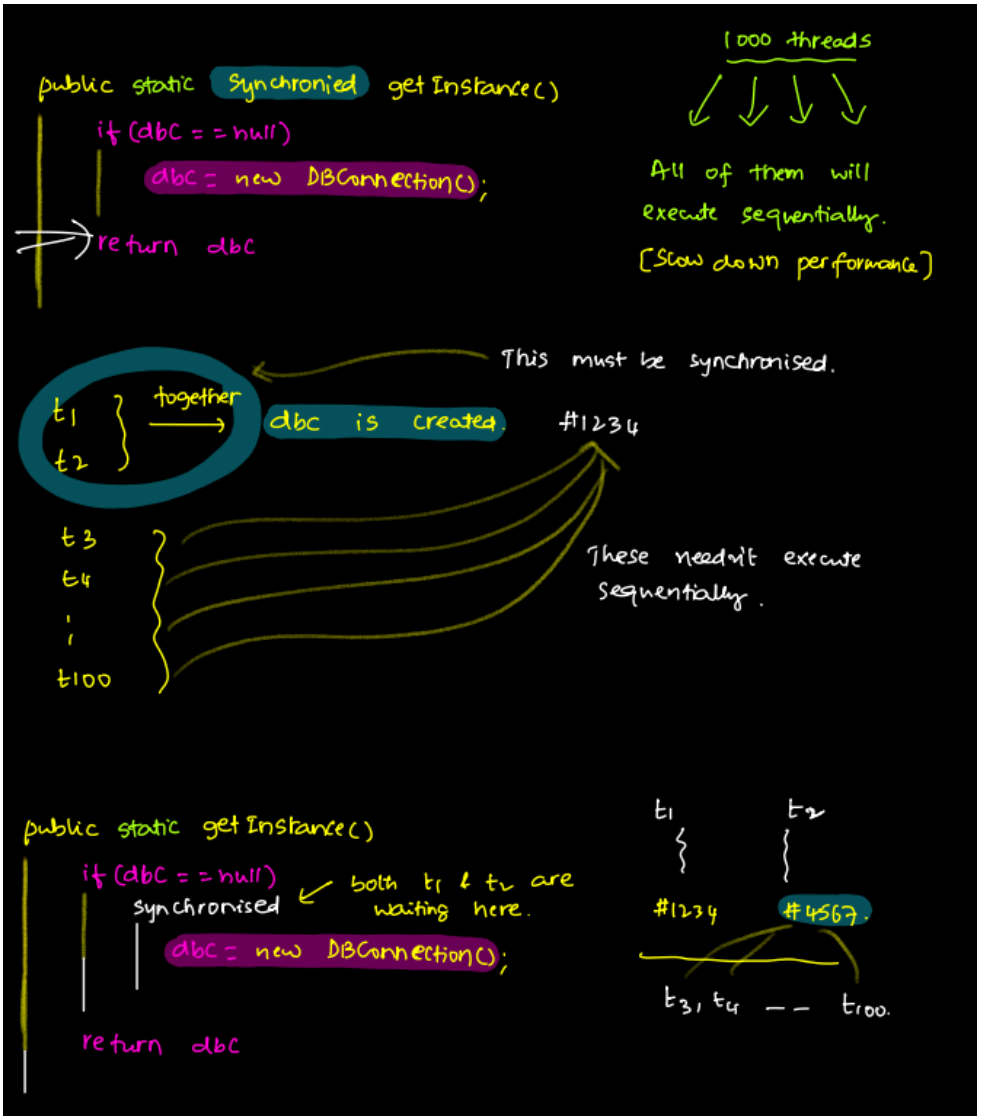


To address the issue with multiple threads in below screen, we can make it synchronized. While this resolves the problem, it has the drawback of being slower, as it ensures that the operations run sequentially.

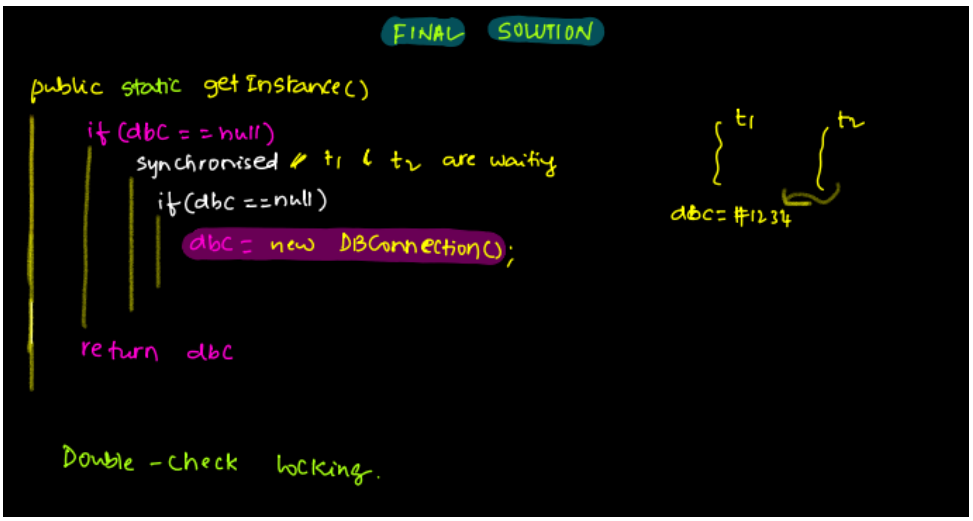




To overcome the sequential problem, we can use a synchronized block instead of a synchronized methodin below screen. However, if two threads arrive simultaneously at the condition, due to synchronization, one thread will enter first, and the other will wait until the first one completes. Once the first thread finishes, the second one will enter the synchronized section and potentially create another instance.



To fix the problem, we can use double-check locking. This means adding an extra condition inside the synchronized part to prevent a second thread from entering at the same time.



Backend LLD & Projects-2: Design Patterns: Builder - Jan 16
https://github.com/kkumarsg/lld/blob/main/src/main/java/designpatterns/builder/App.java

- **Validation before we create an object using constructor**
    - A class with lot of attributes
    - Constructor validations
    - Issues with constructor validation
- **When you can create an object in so many ways.**
    - Constructor overloading
    - is constructor overloading always possible ?
- **Will map solve the issue ?**
    - issues with map
- **A custom data type -> Builder class**
- **Enhancements of Builder class**
  1. getBuilder in student
  2. build method in Builder
- **Modifying the setters a bit to make the builder code neat and use chaining**
- **Final version -> Making constructor of Student class private**
    - Moving Builder inside Student class and make it static inner class
- **What is the benefits of making builder**
- **Why can't we set the objects later, once an instance is created.**
    - Can we always do this ?

a class with lot of attributes.

```
class Student {
        int id
        String name
        int age
        String address
          .
          .
          .
        //20 properties.
}
```

```
Student S1 = new Student()
S1.setId(123)
S1.setName("Tejju");
        .
        .
        .
```

We want to do some validations before we even create an object.

1. gradYear < 2020
2. phNo should be valid.

W/O student object should be created, without these validations.

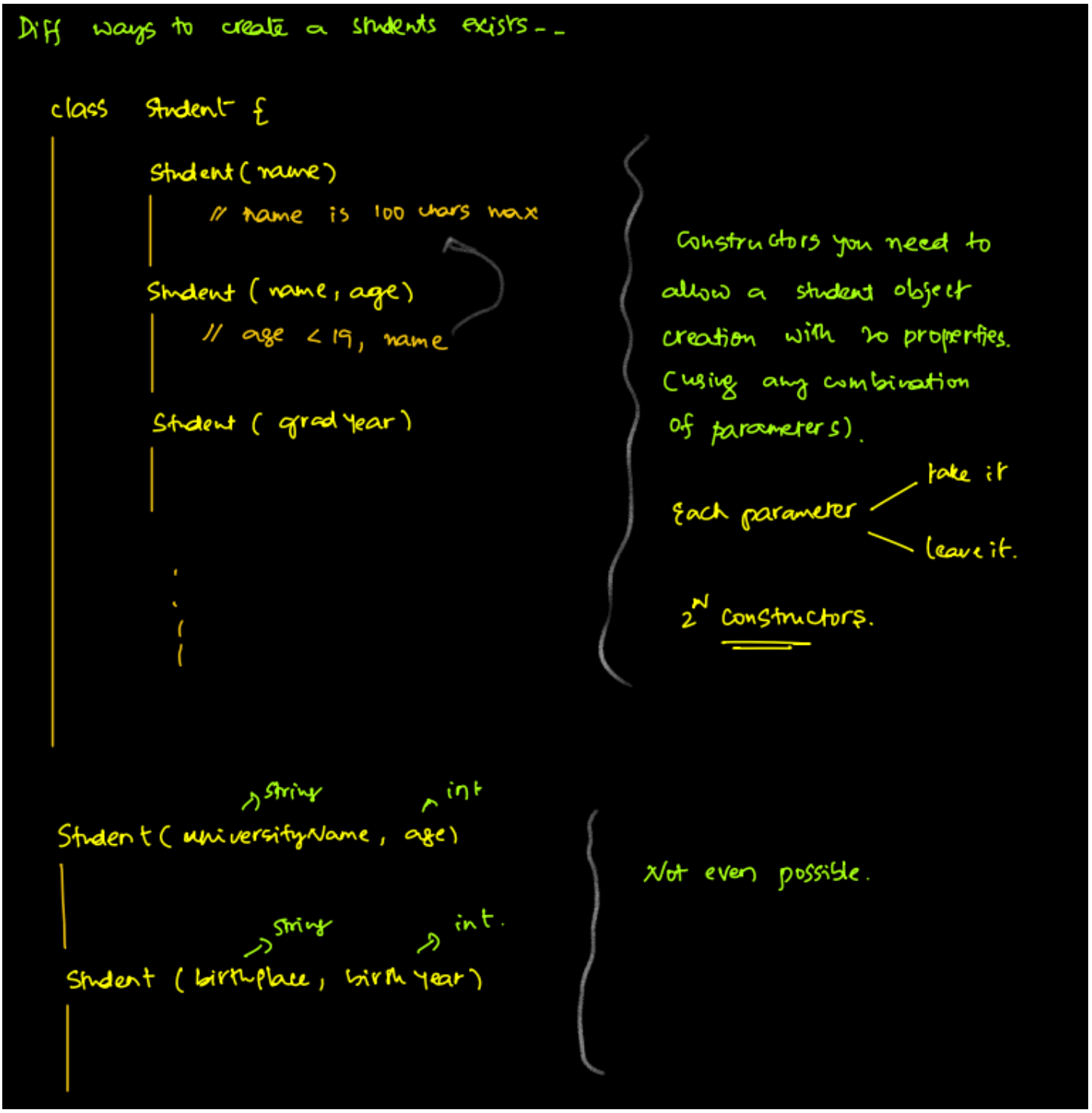
```
Student (int age, int gradYear, long phNo ... )
    1.// validations
      if (gradYear >= 2020)
          throw exception
    2.// ————————
      new Student();

Main()
    St = new Student(12, 2019, 9481482481, ....)
```

Issues

1. order of fields should be maintained.
2. Code is not easy to understand.
3. You want to create an object with some properties.

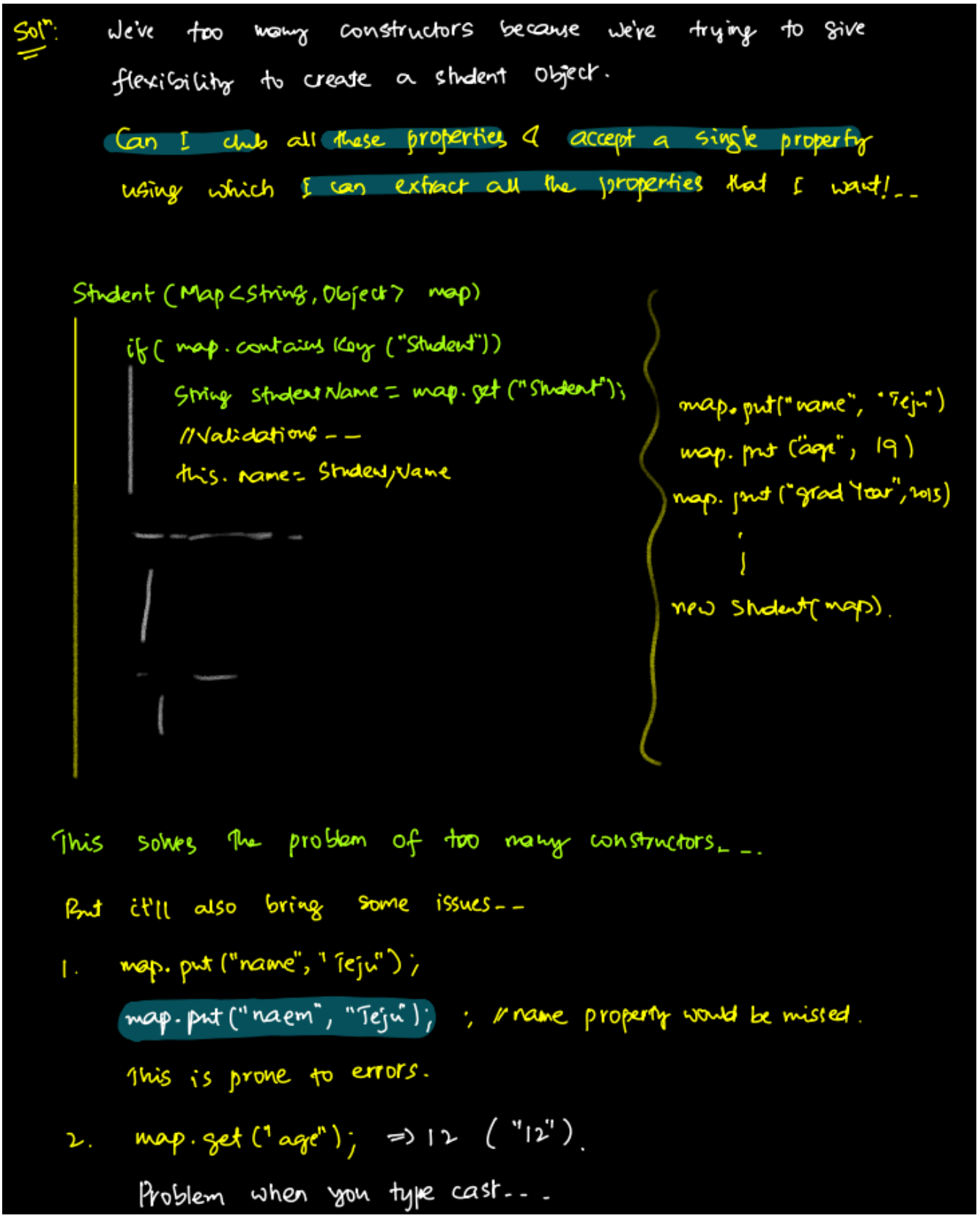
If we attempt to simplify the issue of multiple parameters mentioned above, we will find ourselves creating an exponential number of constructors, approximately 2^n, which is impractical. even if we have two properties with same type it is not possible using constructor It is vary hard to differentiate by complior.

When you can create an object in so many ways

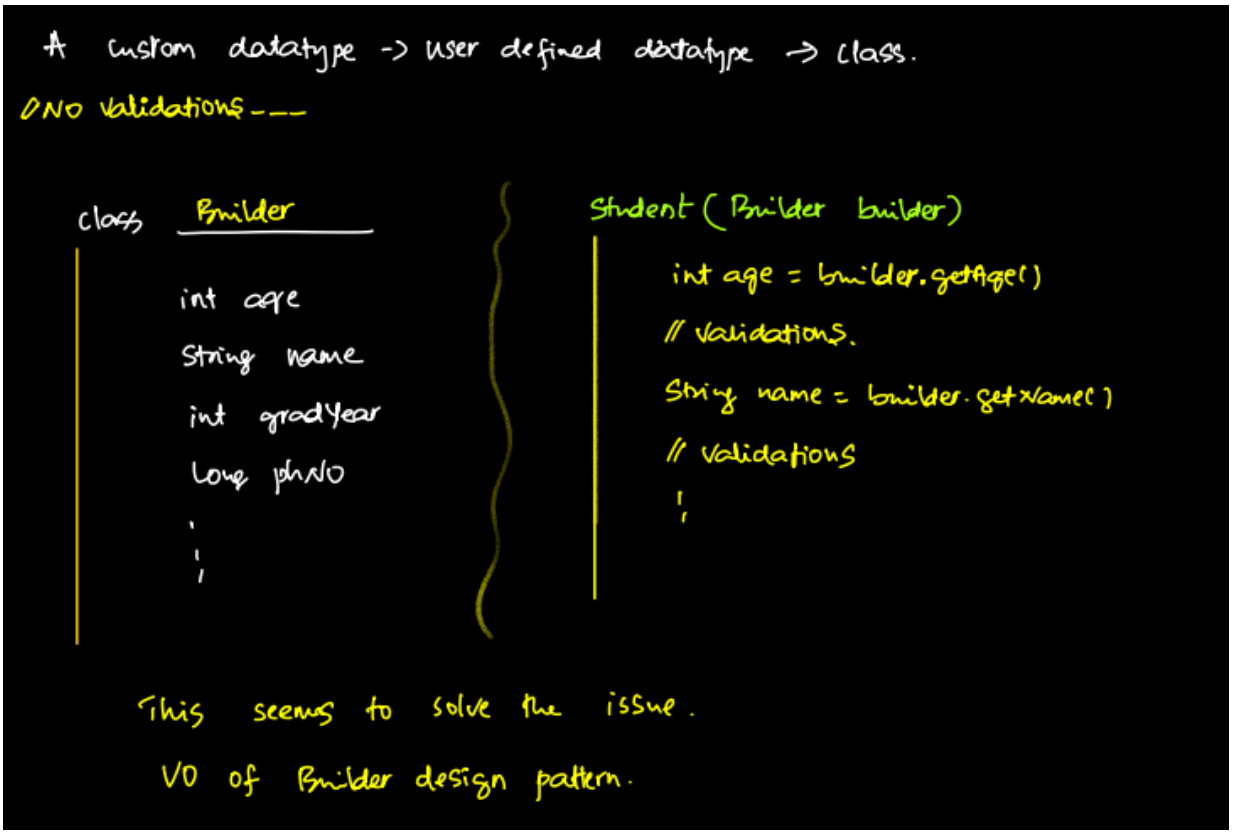


Will map solve the issue ?

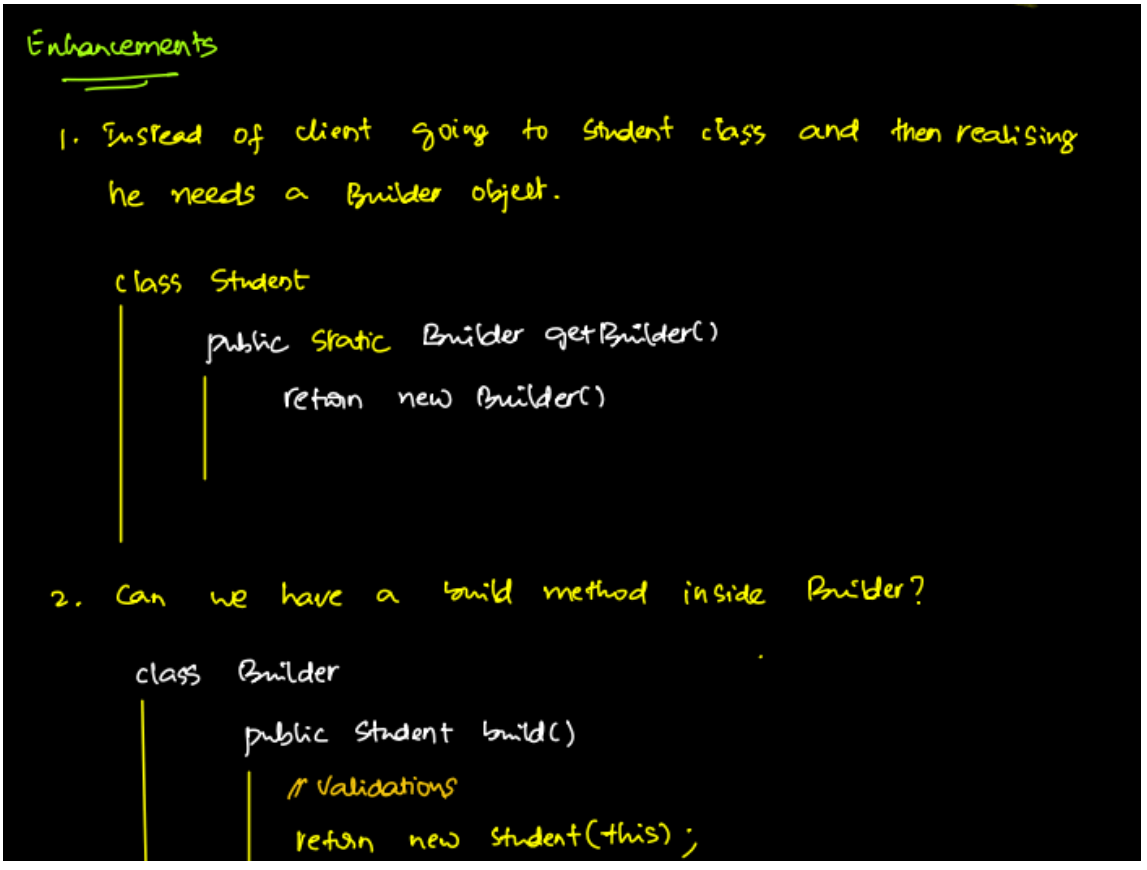
To address the issue of multiple parameters mentioned above, we can explore using a data structure that allows sending multiple properties, such as a Map in the form of key-value pairs. However, using a Map<String, Object> introduces challenges. There is a lack of type safety, allowing the possibility of sending values of any type. Additionally, there is a risk of sending incorrect keys due to the absence of validation or information about the expected key types, leading to potential mistyping errors



A custom data type -> Builder class

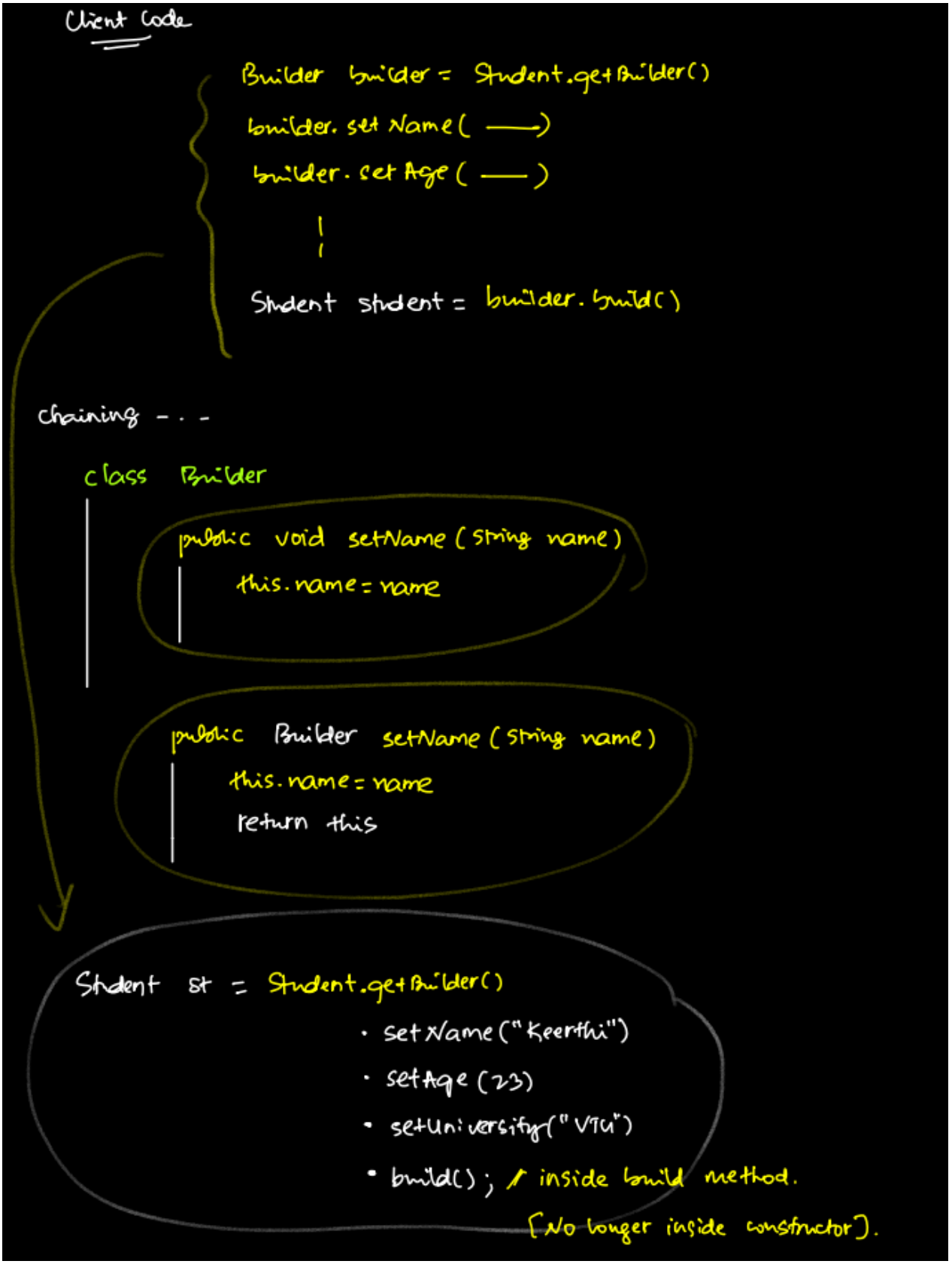


Enhancements of Builder class -> As observed, creating a student object requires visiting the builder class each time, which seems unnecessary. To address this, we have two approaches. The first is to create a getBuilder method. A better option, however, is to instantiate the student object within the builder class and return it to the client after validation.

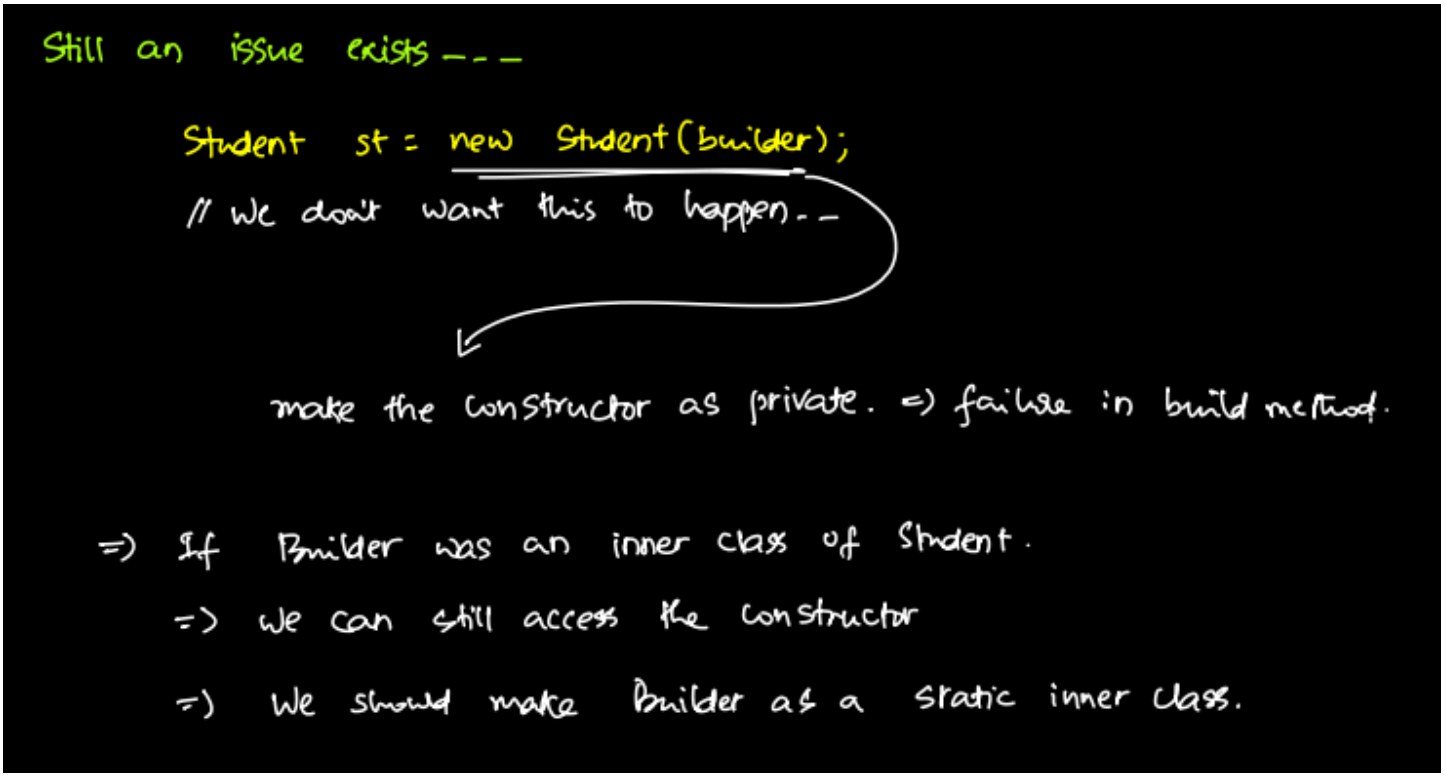


Modifying the setters a bit to make the builder code neat and use chaining

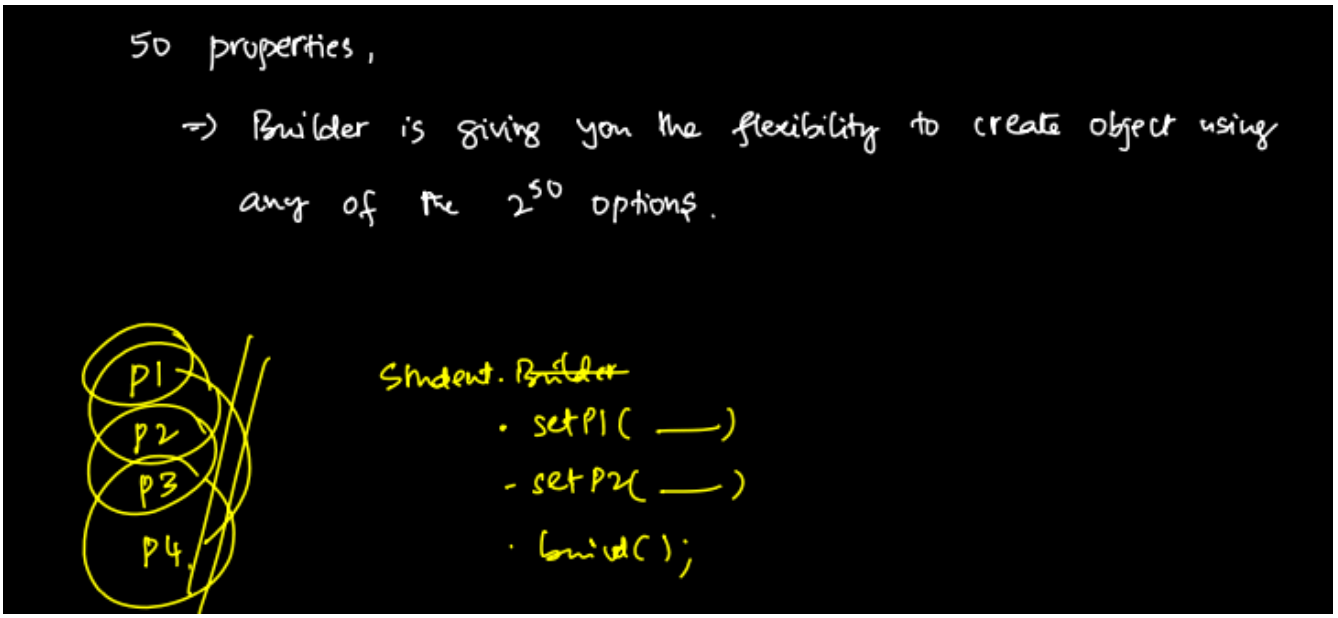
In the client code below, we can observe that for the aforementioned enhancement, we need to create a builder object and make multiple calls to set properties. To address this, a minor adjustment can be made to the setter methods in the return type. Instead of returning void, we can change it to return the builder. This modification allows us to chain all setter methods using a fluent interface, as each setter method now returns the builder.



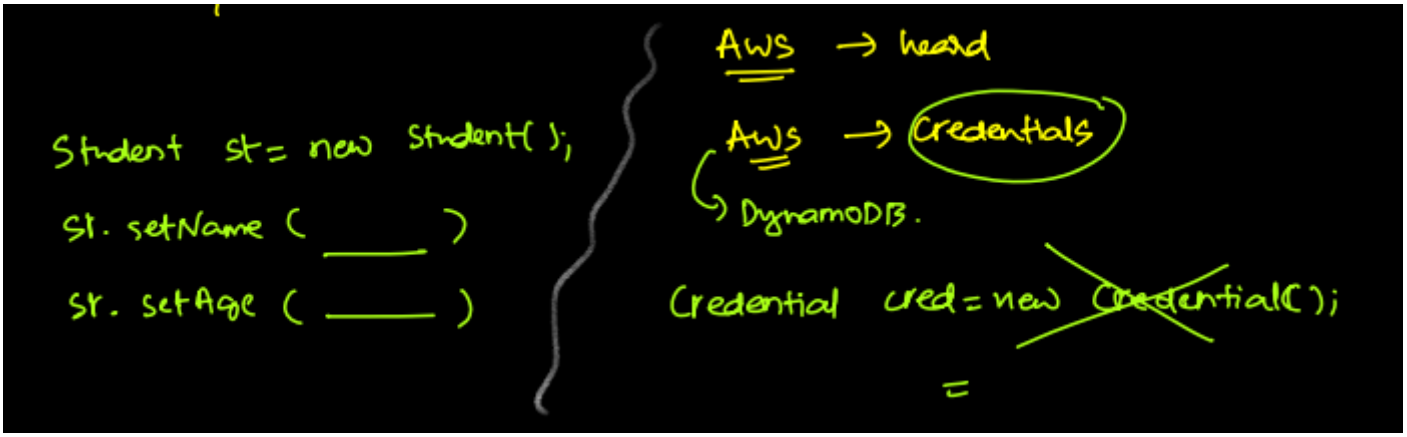
Final version - Making constructor of Student class private -> The solution above appears to be optimal, but there is another issue. Since the student class is public, anyone can still access it directly for creation. To address this, we need to make the student constructor private. However, converting the student constructor to private would make it inaccessible for the builder class as well. To resolve this issue, we will move the builder class inside the student class as an inner class and make it static to allow calling without an object.



What is the benefits of making builder -> Creating a builder provides the benefit of being able to construct objects with only the necessary properties. This approach allows us to avoid the complexity of having 2^50 constructor options for 50 properties and resolves the problem associated with constructor overloading.



Why can't we set the objects later, once an instance is created -> In certain scenarios, there is a requirement to restrict access to modifying an object after its creation. For instance, when accessing DynamoDB in AWS, credentials need to be provided. However, it is not advisable to create new credentials by instantiating a new object using the credentials class. To address this concern, the builder class emerges as the optimal solution.



Backend LLD & Projects-2: Prototype and Registry - Jan 18

- **What does copy an object mean here ?**
  - Basic solution
  - Issues in basic solution
- **Introducing Prototype design pattern and fix for a problem**
- **More intuition for prototypes**
- **How to store an object -> Registry**
- **Prototype and Registry difference**
- **Creating Prototype interface**

What does copy an object mean here ? and Issues in basic solution

As indicated in the displayed screen, directly assigning obj1 = obj2 only duplicates the reference, not the actual object values. To replicate values, a manual copying process is required for each object value. However, this method presents challenges, including the necessity to be aware of all object properties. Furthermore, the object may contain private fields inaccessible from the client class, complicating the identification of the object being copied. Another complication arises when dealing with subclasses of the student class. If code is written for the parent class, defining a type for instances of both the student and intelligentStudent, it becomes challenging to discern how instances are created for either the parent or subclass, as explained in the third issue. Addressing these concerns requires the implementation of multiple conditions, potentially violating the Single Responsibility Principle.

Understanding copy

```
Student st = new Student();
Student st1 = st;
```

creating a new object.

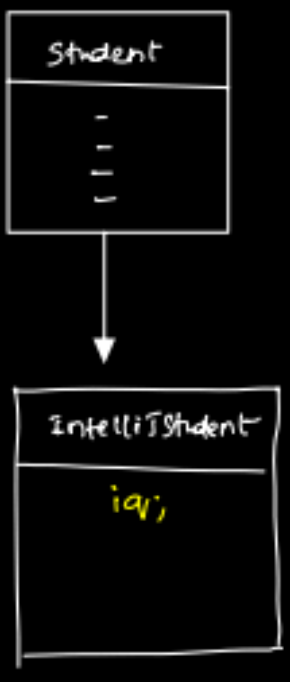
Not a copy, we're ref to same object.

Basic sol^n: Create a new object and copy every single field.

```
Student s = _____
Student sCopy = new Student()
    sCopy.id = s.id
    sCopy.name = s.name
      :
      :
```

Issues with this approach

1. We need to know all the details about student class.
2. Some fields could be private.
3. .



passed from somewhere.

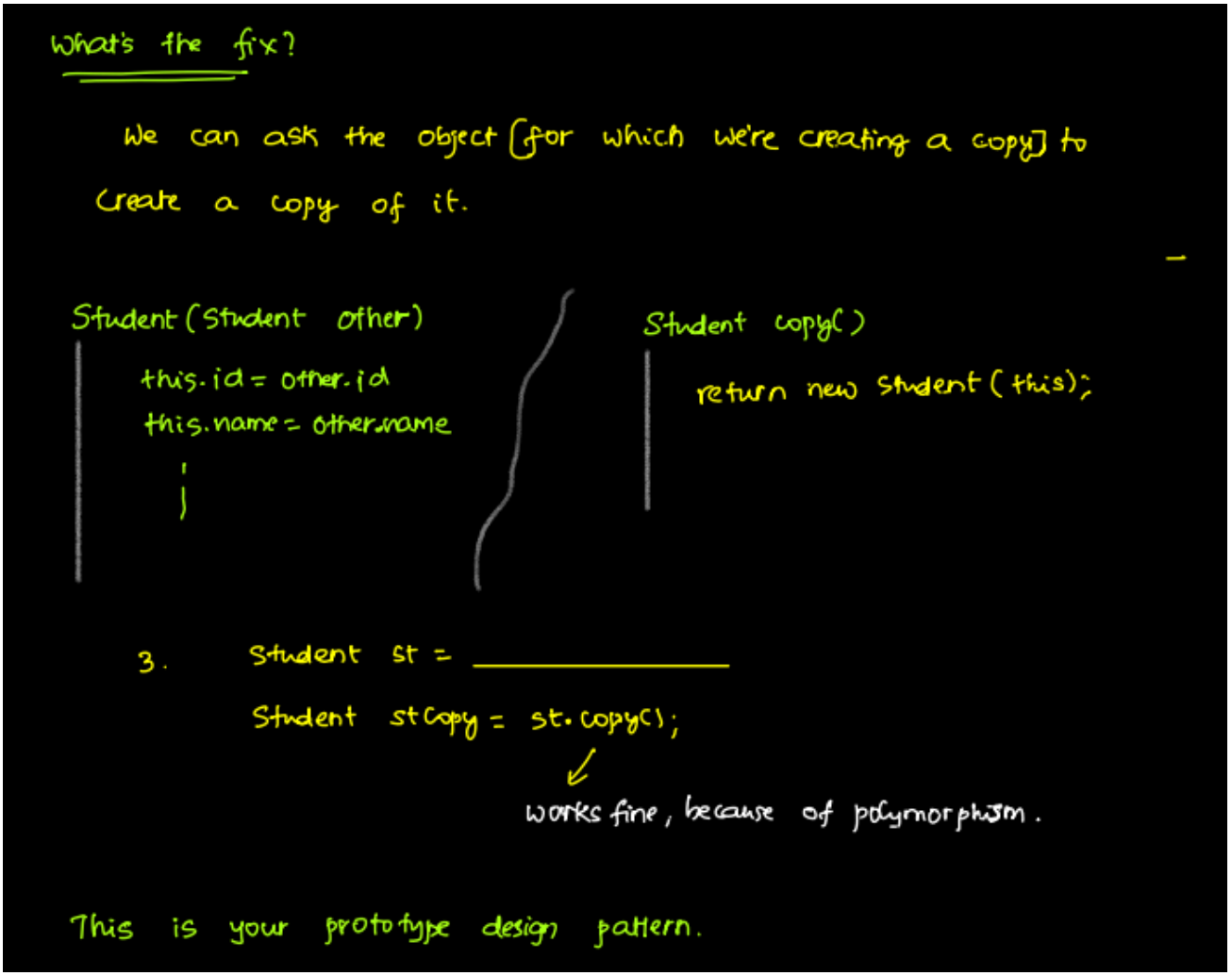
```
Student st = _____
```

```
if(type of st == IntelliJStudent)
    _____
    _____
    _____
else if (type of st == Student)
    _____
    _____
    _____
```

Violates SRP, OCP.

Introducing Prototype design pattern and fix for a problem

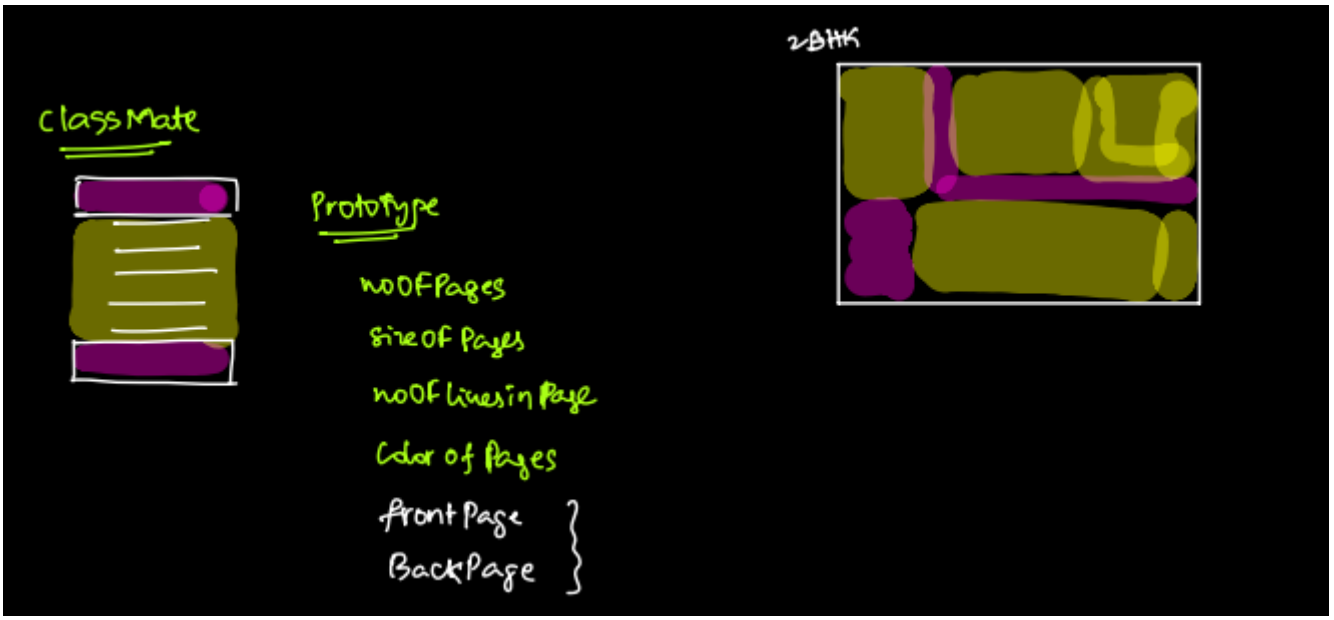
The Prototype design pattern is a creational pattern that focuses on creating objects by cloning an existing object, known as the prototype. This pattern is useful when the cost of creating an object is more expensive or complex than copying an existing one. the prototype design pattern is typically implemented using the Cloneable interface



As observed in the above display, issues arise when copying objects from the client class. To address this, we can implement prototype design pattern by creating a copy constructor and a clone method within the same class as the object. The class, being more aware of the object's structure, can effectively handle all three previously discussed problems. Initially, we create an interface, implementing it in all relevant classes, including the parent and subclasses. Each class can override the methods based on its specific requirements, leveraging polymorphism to ensure an efficient copying process that accommodates the instance object.

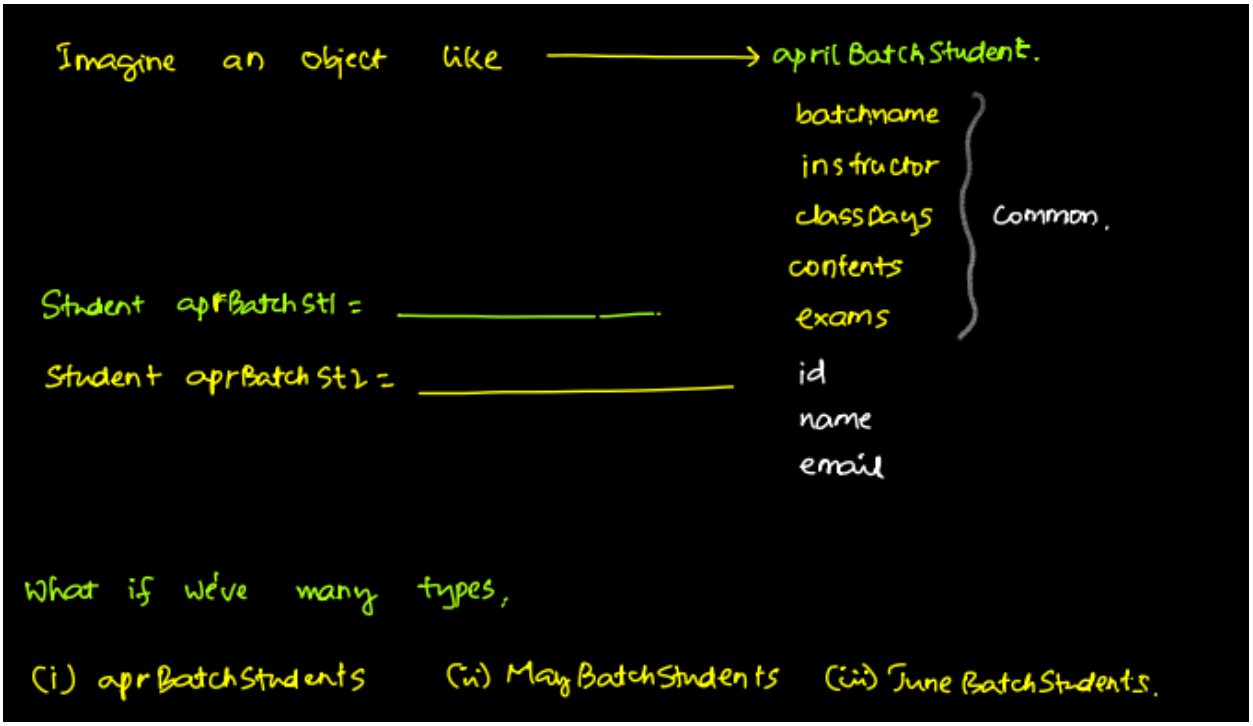
More intuition for prototypes

In simple terms, a prototype consists of common fields shared across all classes, with specific fields unique to each class. In the example below, the 2BHK green blocks represent the prototype, containing shared features present in all flats. This serves as a common blueprint or prototype. The purple blocks represent custom requirements, acting as custom fields specific to subclasses. Subclasses inherit the prototype (green) and can add custom elements. Additionally, each subclass may create a copy method tailored to its specific instance.

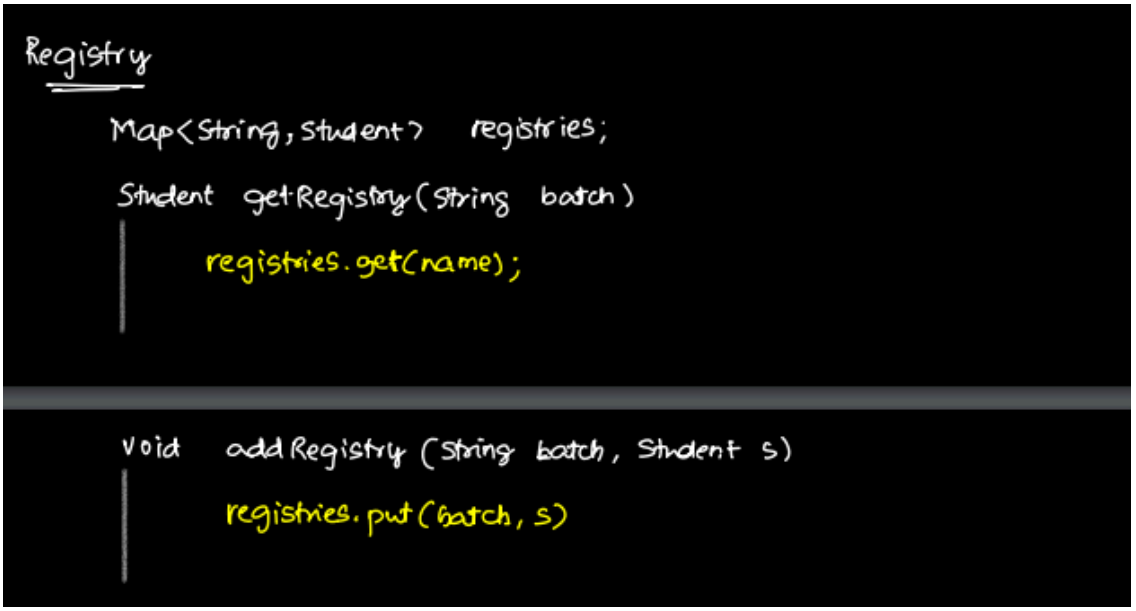


What is Registry ->

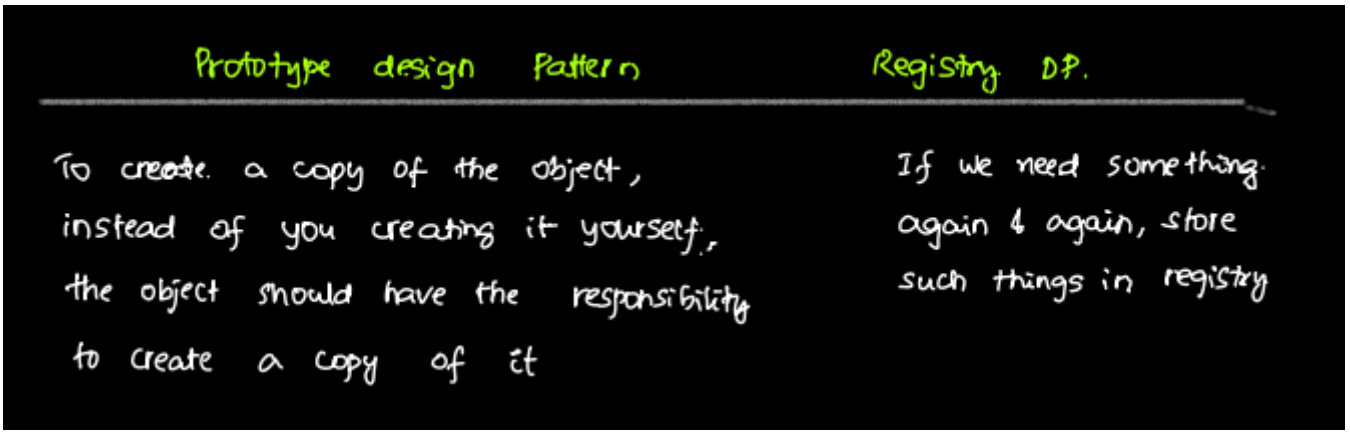
Centralize the management of instances or services to provide a global point of access. Decouple the components of a system by allowing them to interact through a registry instead of directly with each other.



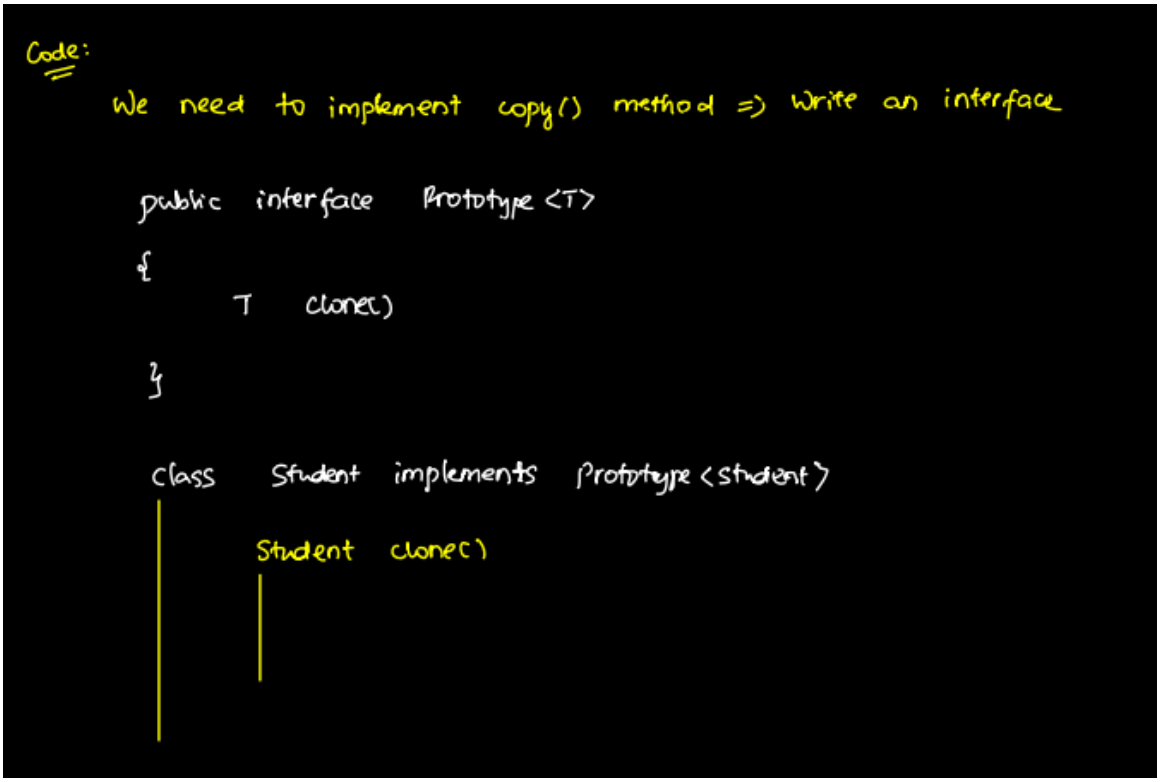
How to store an object -> Registry



Prototype and Registry difference

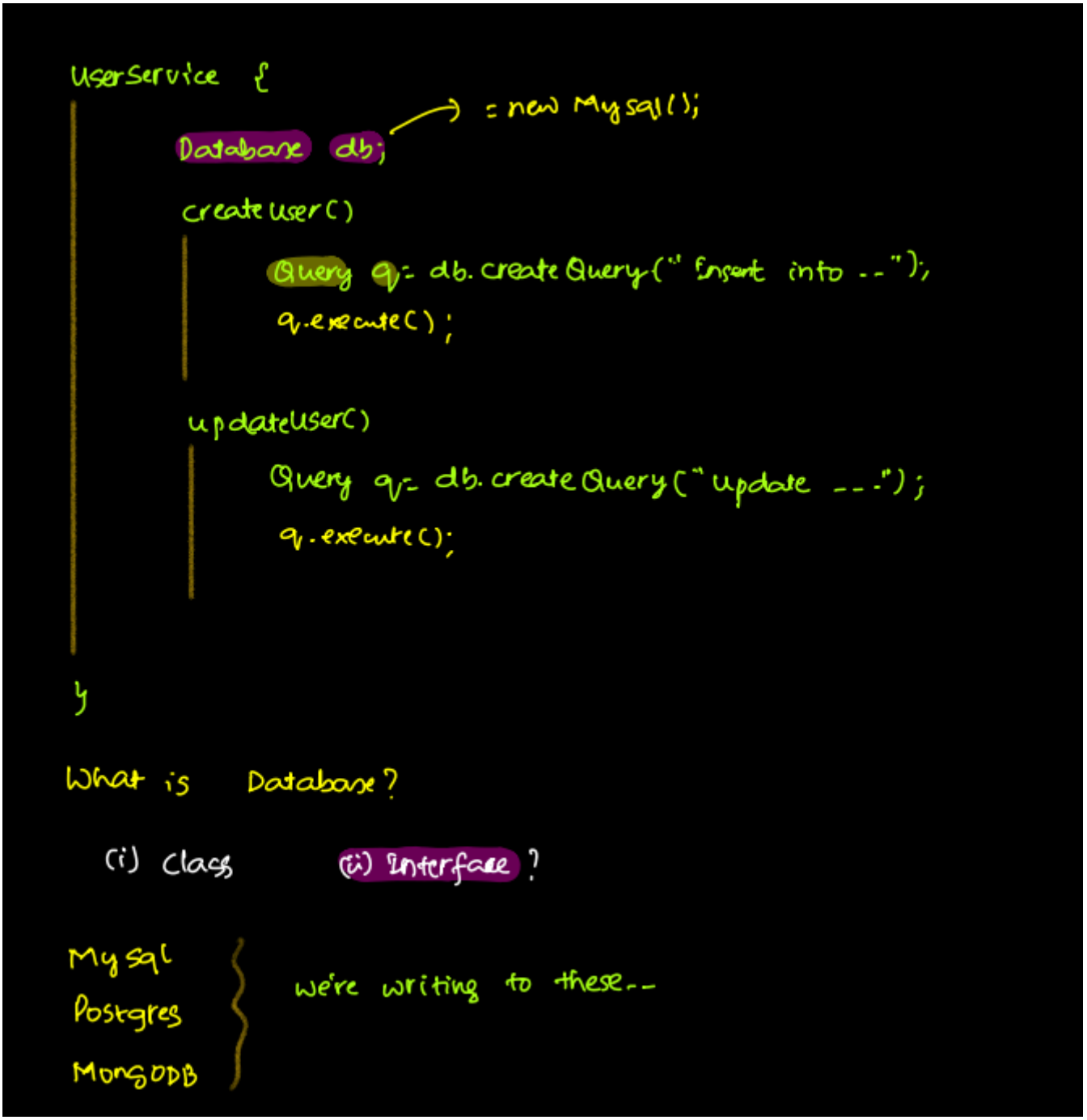


Creating Prototype interface



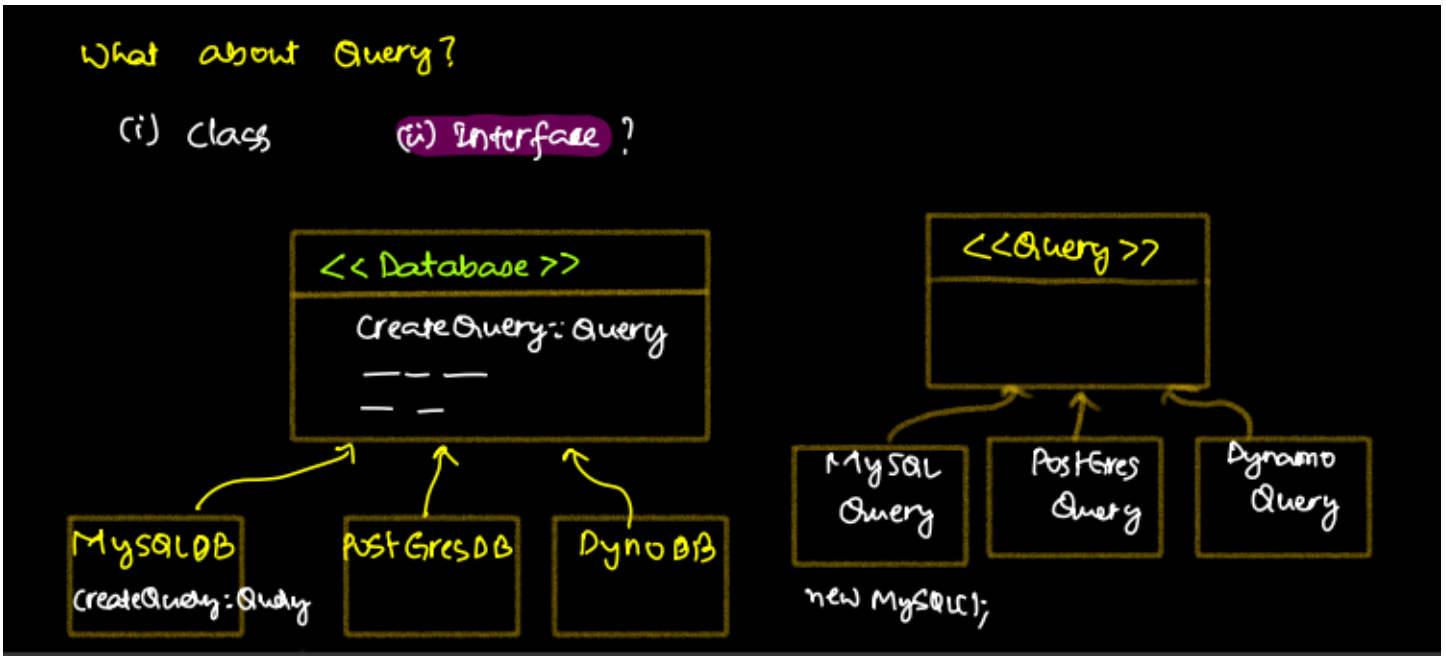
Backend LLD & Projects-2: Design Patterns: Factory - Jan 20

- **Why database should be interface**
- **Should Query be also an interface ?**
- **Factory method**
- **Introducing abstract factory**
- **Practical example of Factory for Flutter**

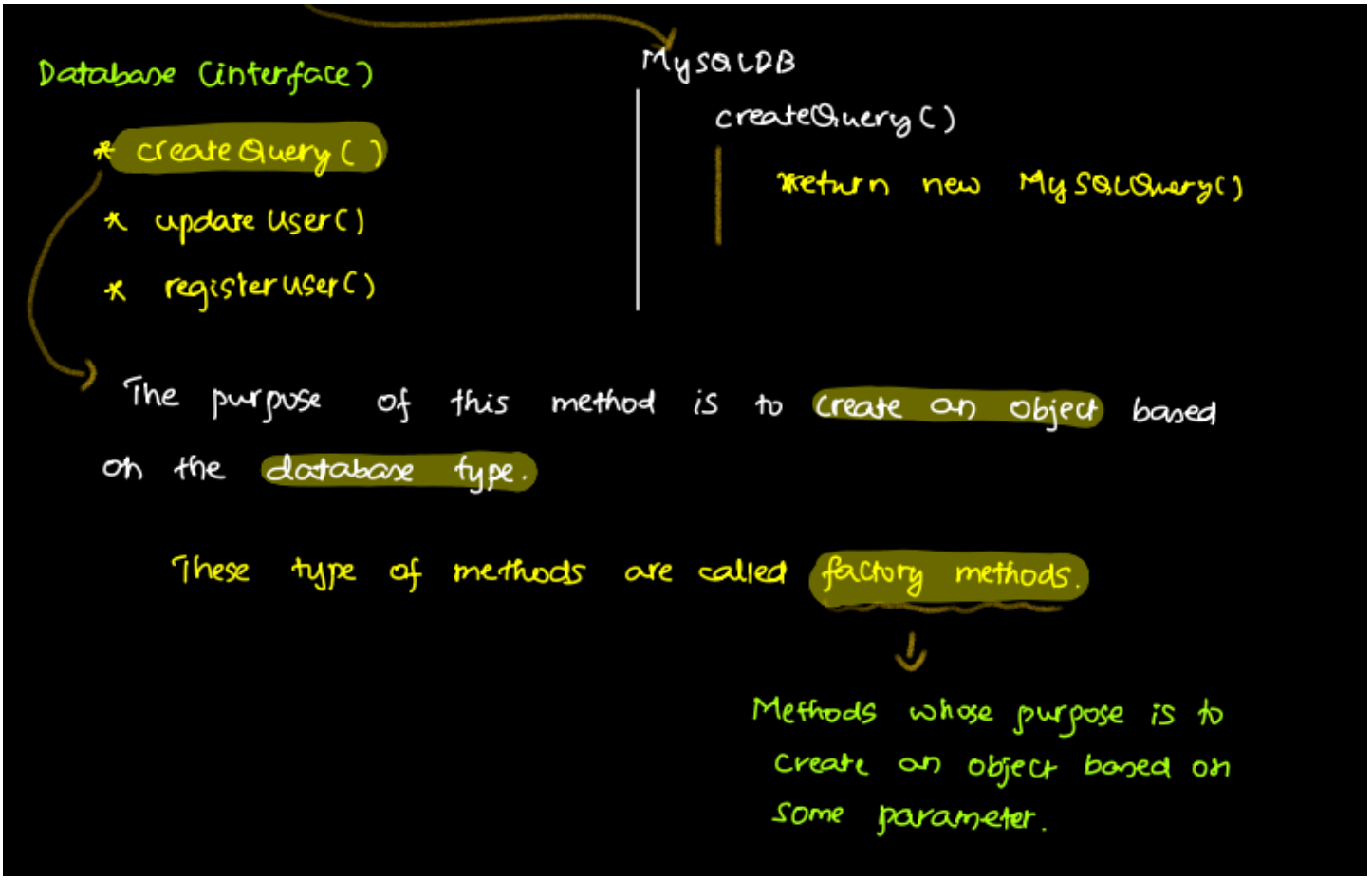


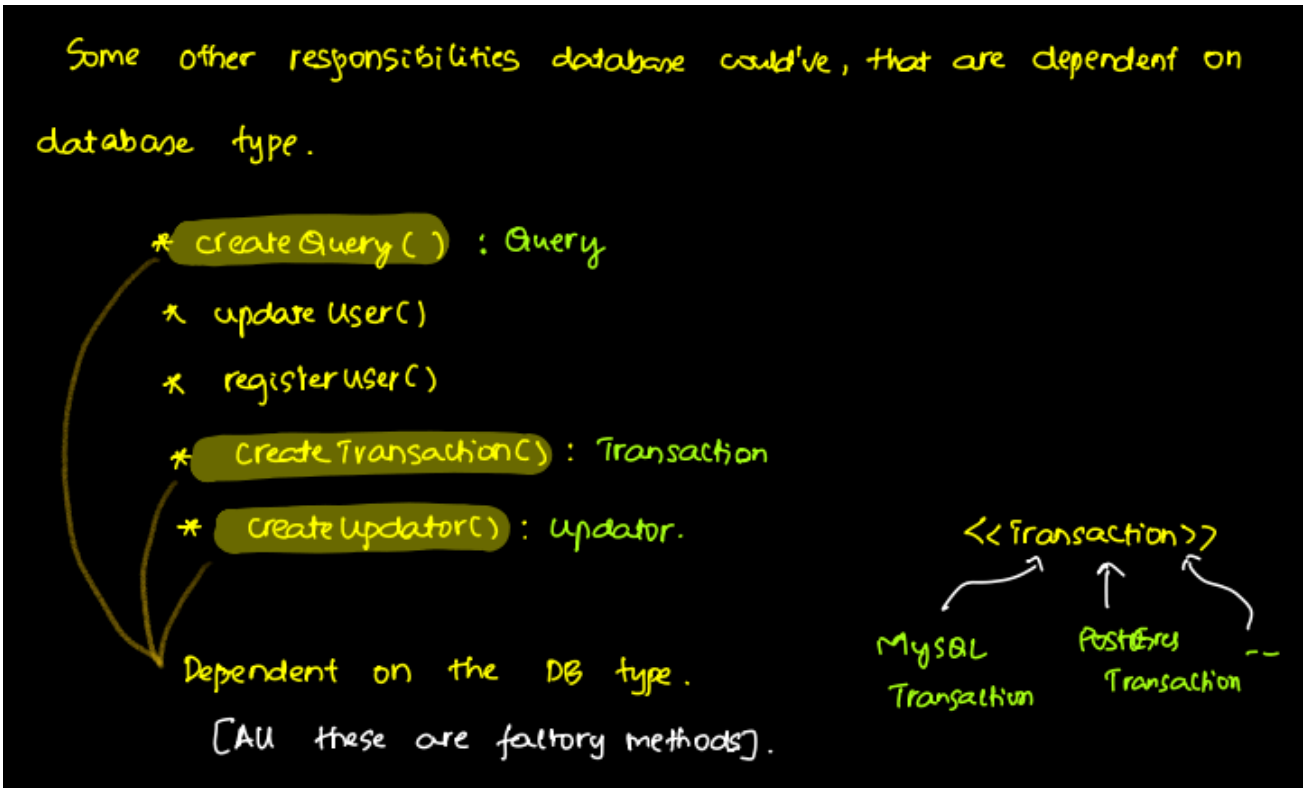
Database is a interface if it is the class then it will break the solid rule **Dependency Inversion Principle (DIP) ->** The Dependency Inversion Principle suggests that high-level modules should not depend on low-level modules; both should depend on abstractions. By defining a database interface, you adhere to this principle, making your codebase more maintainable and adhering to best practices in object-oriented design.

Should Query be also an interface ?
Certainly, the query is also an interface, as it serves as a common standard implemented by various types of databases such as MySQL, PostgreSQL, and DynamoDB.

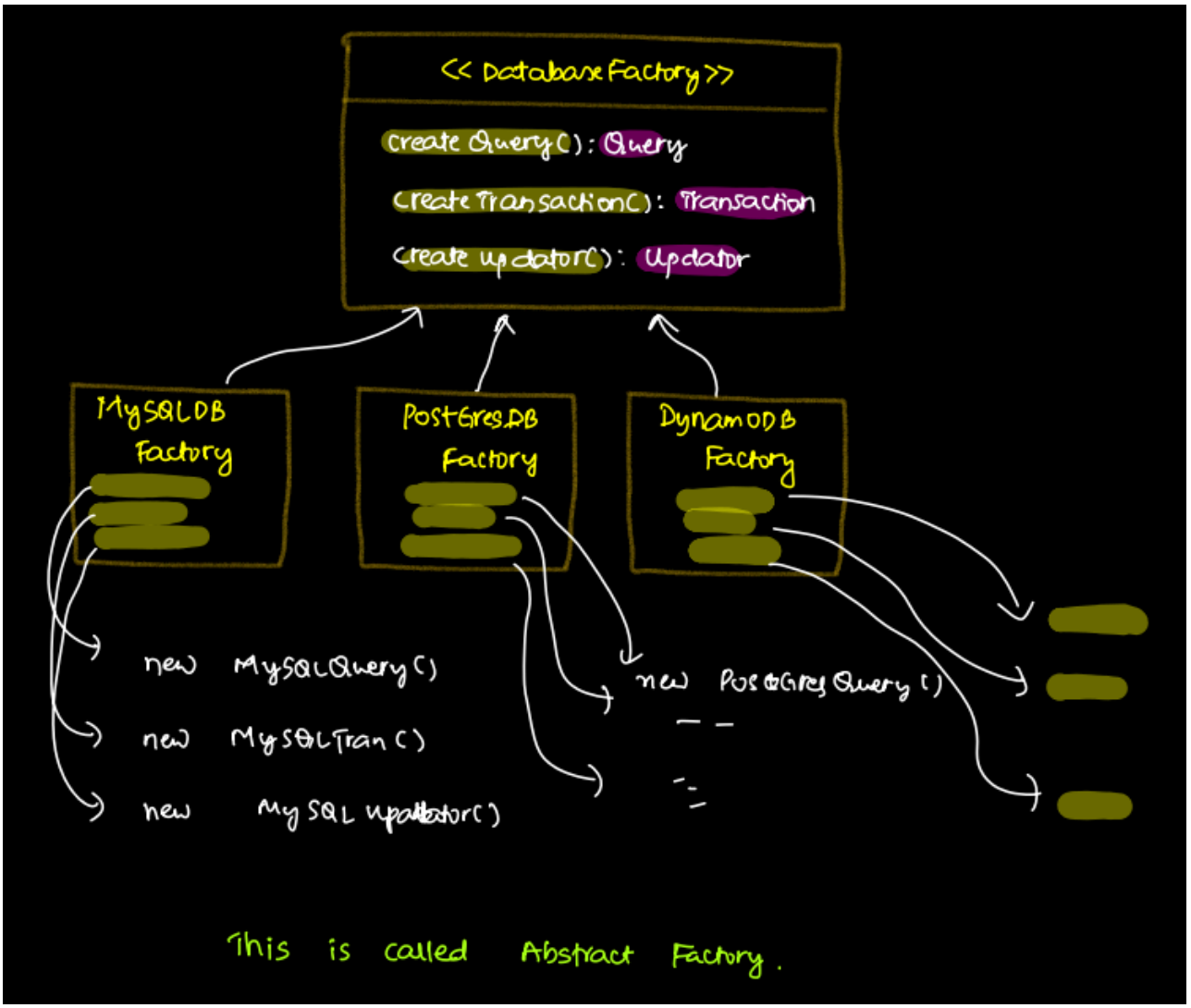


Factory method -> the Factory Method is a creational design pattern that provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created. This pattern promotes loose coupling by abstracting the object creation process.

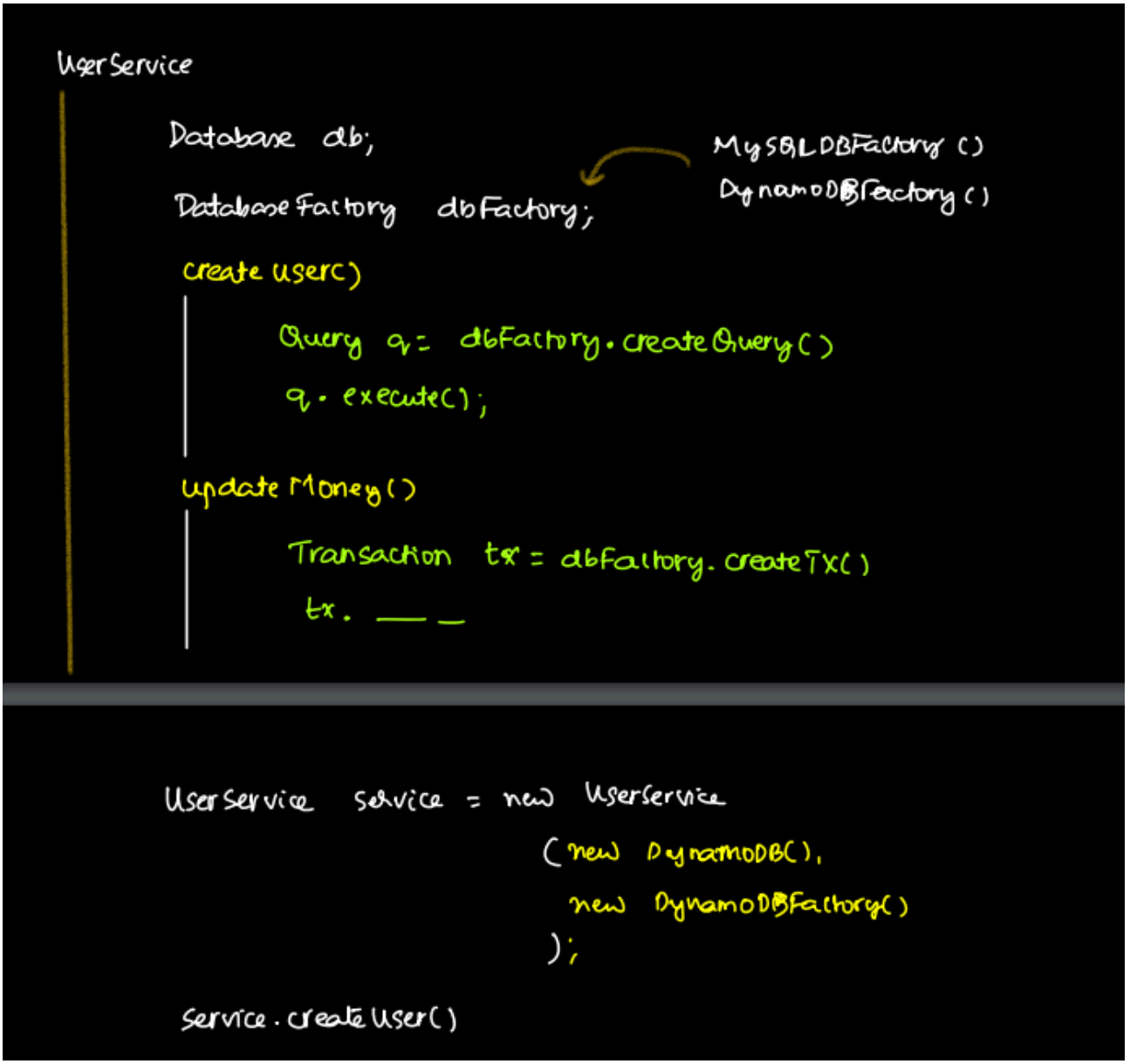




We observe that these versatile methods, collectively referred to as factory methods, cater to multiple classes. They operate by returning objects based on different database types.



These methods are doing similar tasks for various types of databases. To simplify, we gathered them into one abstract factory. Now, this factory can give you the right tools (like queries or transactions) based on the type of database you're working with.



Step1: Create a Abstract Factory Interface (Include all related factory methods);
Step2: Create a service file create db and factory fields assign this using constructor passed by instance.
step3: Create respective database classes implements factory interface.
Step4: Create Instance and pass required class instance

Practical example of Factory for Flutter
https://github.com/kkumarsg/lld/blob/main/src/main/java/designpatterns/factory/Flutter.java

Imagine a Flutter class that provides button and menu UI components for various platforms. Implementing this functionality would typically involve creating multiple conditions based on different platforms, potentially violating the solid open-close principle. To address this, we can employ a factory method. This method will return the appropriate button based on the platform type, and we'll establish an interface for the button, which will be implemented by the factory method. We'll apply the same logic for the menu as well.

Create Fluter UIFactory

Step 1: Instantiate a Flutter class and declare a field named UIFactory. Create constructor and assign UIFactory using passed instance

Step 2: Develop an interface named UIFactory with two abstract methods: createButton of type Button interface and createMenu of type Menu interface.

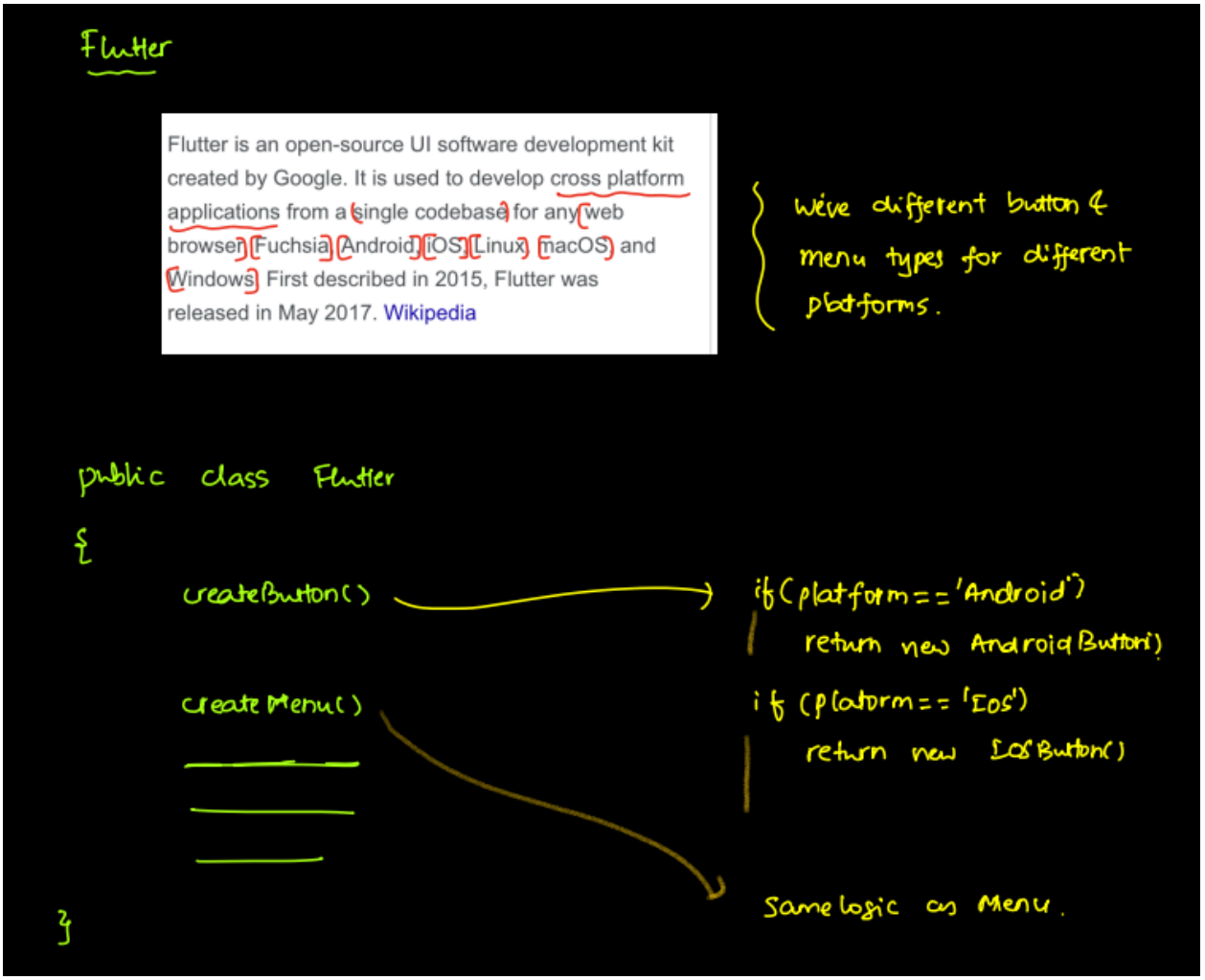
Step 3: Create the Button and Menu interfaces along with their reuired methods createMenu and createButton.

Step 4: Create a pageLayout method in the Flutter class. Within this method, instantiate a button of type Button interface and call the createButton method from the UIFactory interface. Repeat the process for the menu by calling the createMenu method.

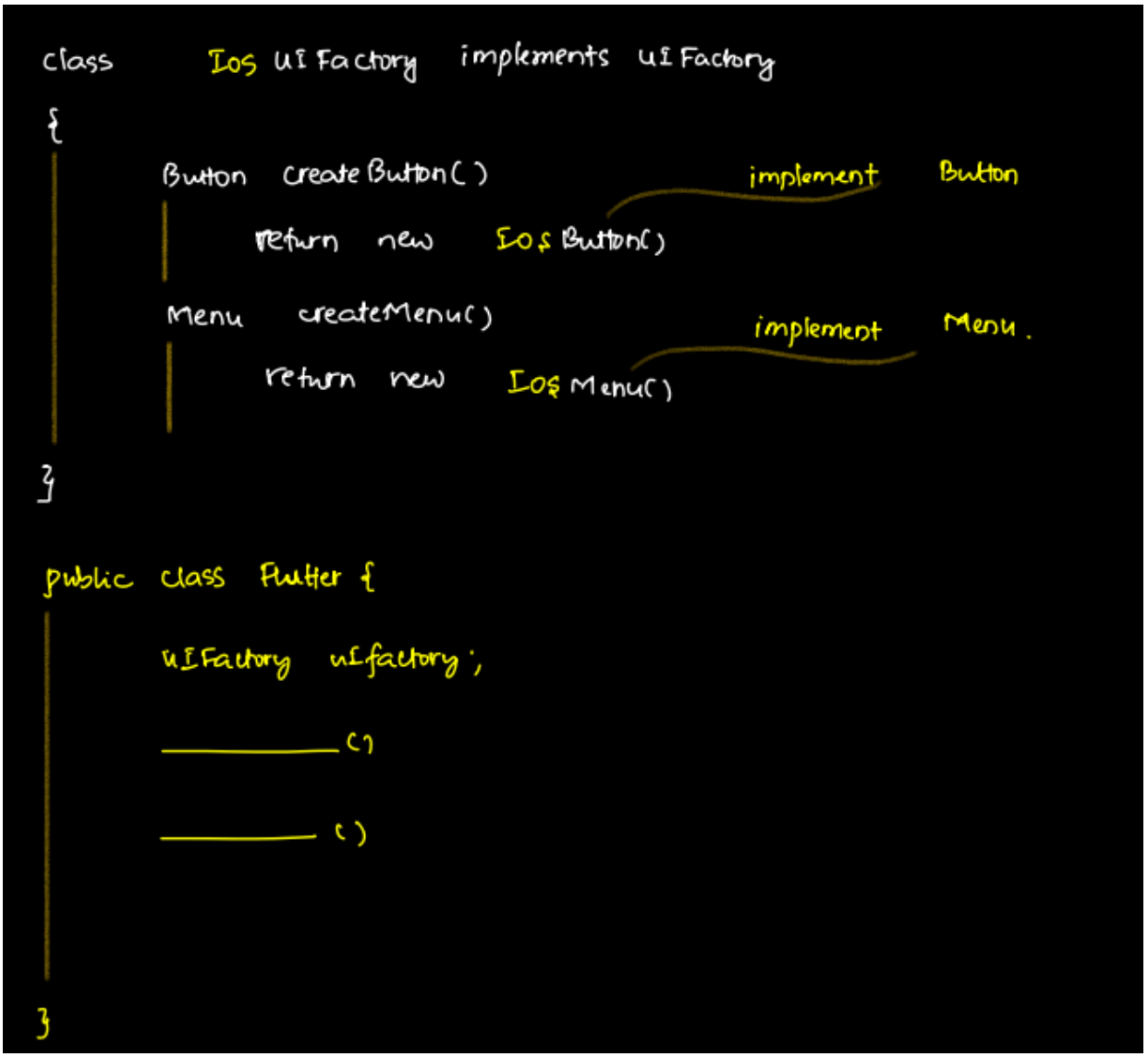
Step 5: Organize the code into a package for Android. Create two classes within this package: one for the button and another for the menu. Both classes should implement UIFactory interfaces and provide the necessary implementations and return instance.

Step 6: Develop an App class and a Main method. Inside the Main method, instantiate a Flutter object. Pass instances of both AndroidUIFactory and IosUIFactory to the Flutter object, and then invoke the pageLayout method like below.

```
Flutter flutter = new Flutter(new AndroidUIFactory());
flutter.pageLayout();
```



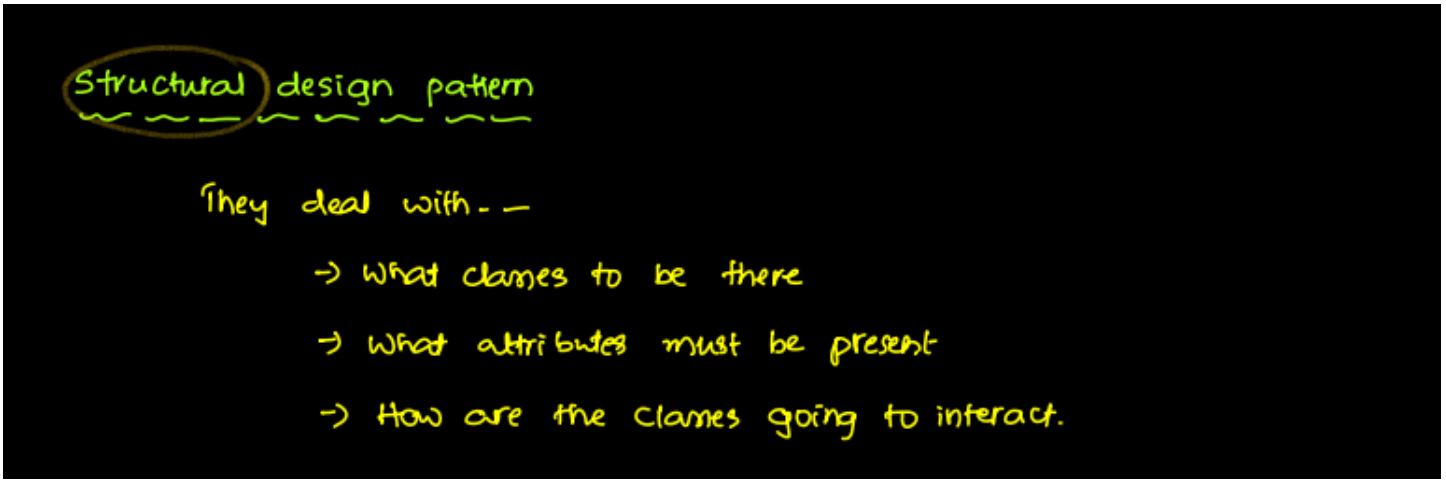

public class Flutter {
_____()
_____()
public UIFactory createFactory() {
}
}
interface UIFactory {
Button createButton()
Menu createMenu()
interface Button {
}
interface Menu {
}
class AndroidUIFactory implements UIFactory
{
Button createButton()
return new AndroidButton()
Menu createMenu()
return new AndroidMenu()
}
implements
implements

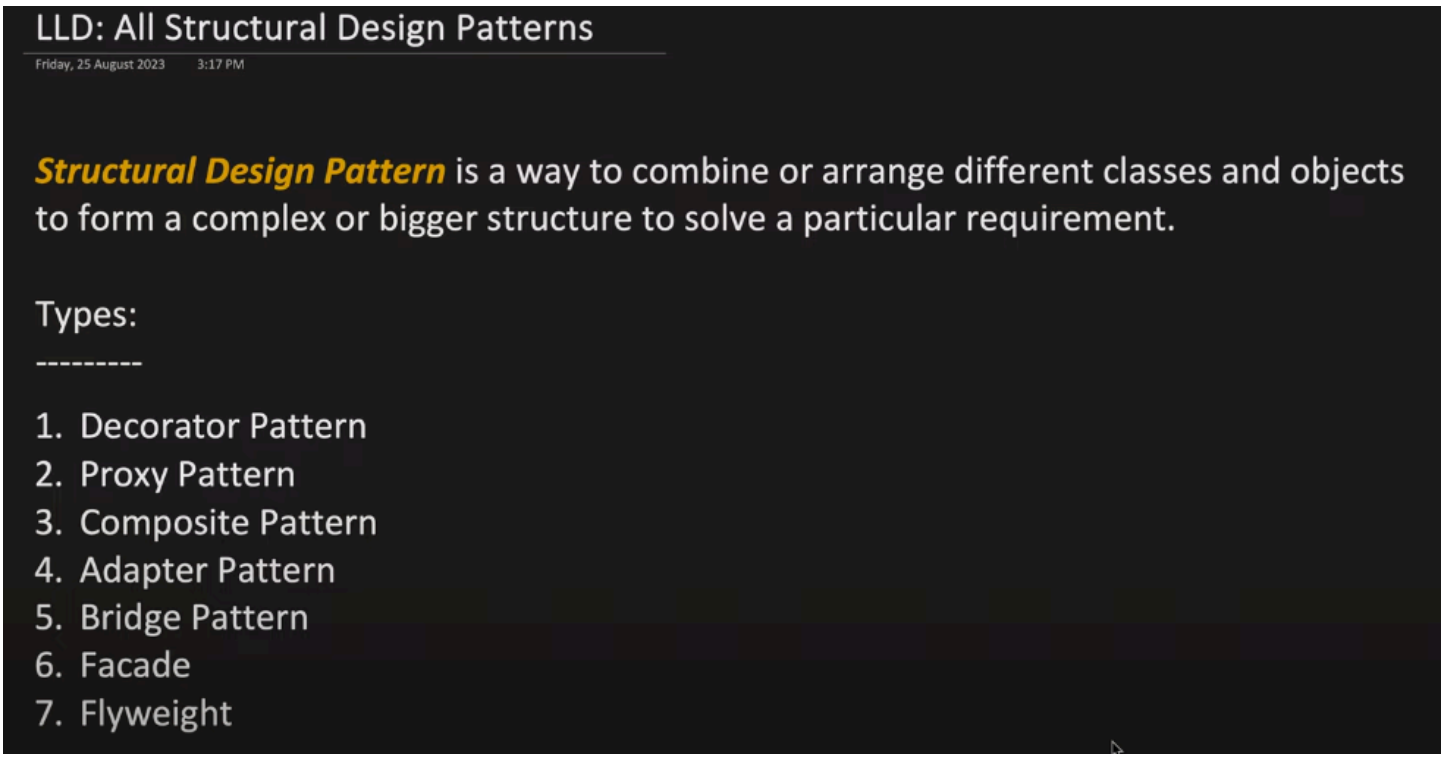


Backend LLD & Projects-2: Adapter and Facade Design Pattern - Jan 25

- **Structural design patterns**
  - **Adapter : What is an adapter ?**
    - **Definition of adapter**
    - **Let's build phonePe from the scratch**
    - **What are the problems in this phonePe design ?**
    - **What is the solution ? Interface ?**
    - **Problem with interface and how adapter solves it**
    - **Creating adapters starts here**
  - **Facade : What is an Facade design pattern ?**

Structural design patterns -> Structural design patterns are design patterns that focus on simplifying the organization of classes and objects to form larger structures. They help in achieving better code organization, flexibility, and maintainability.

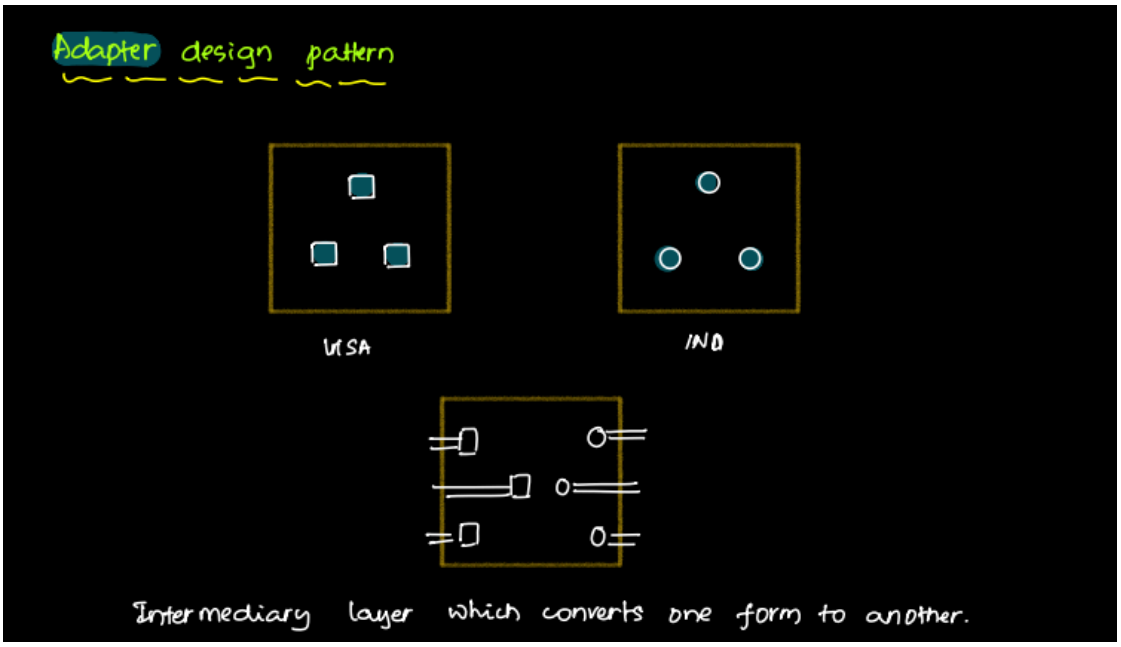




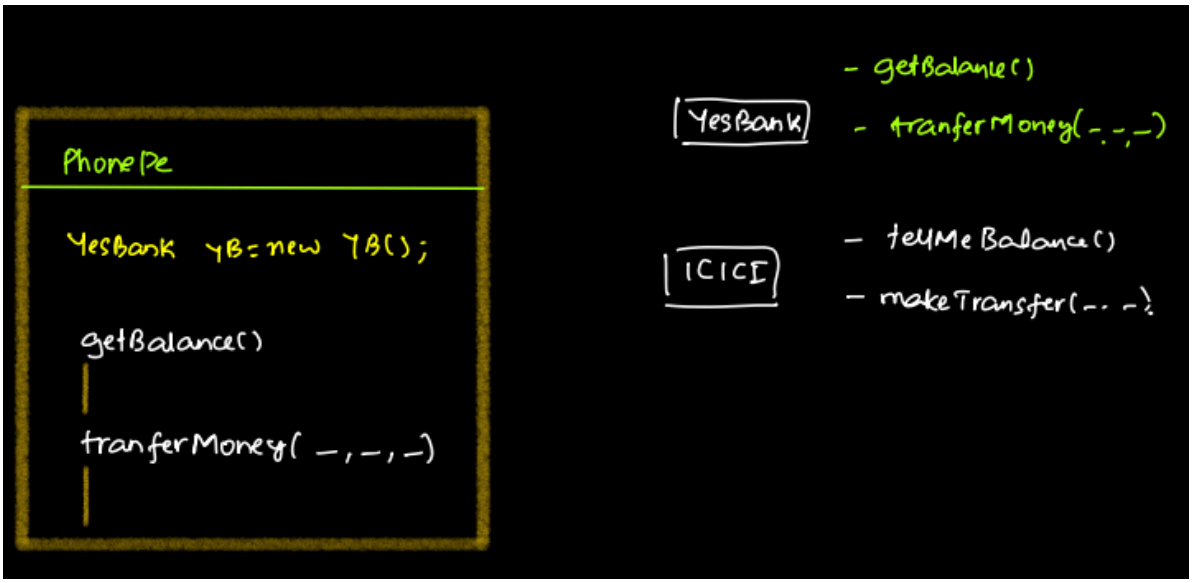
• **Adapter Pattern - cover** • Bridge Pattern • Composite Pattern • **Decorator Pattern - cover**	• **Facade Pattern - cover** • **Flyweight Pattern - cover** • Proxy Pattern • Composite Pattern

Adapter Pattern -> Purpose: Allows the interface of an existing class to be used as another interface. **Intermediary layer which controls one form to another.**
Example: java.util.Arrays.asList() method that allows an array to be treated as a list.

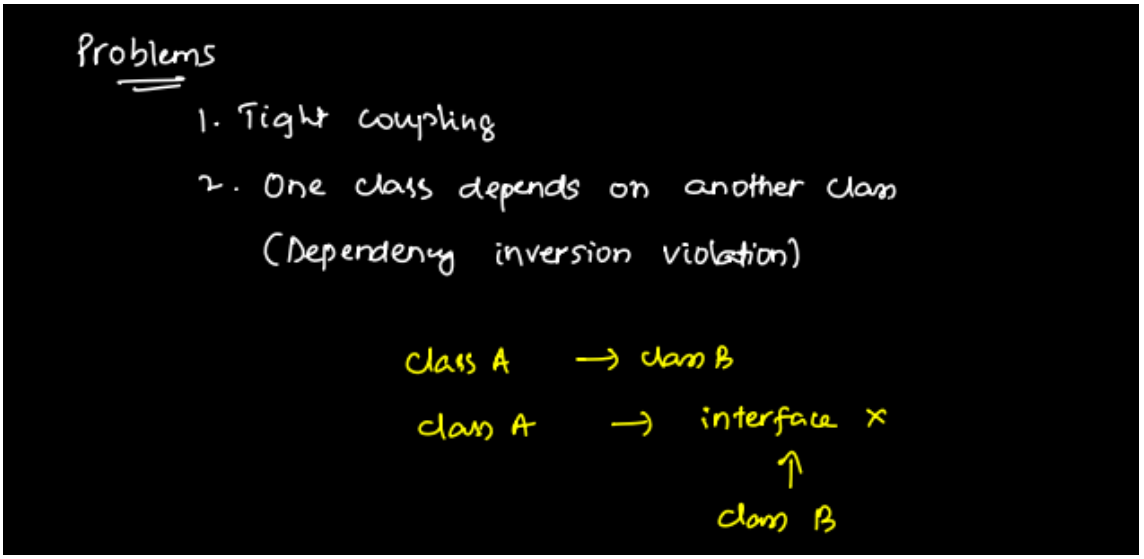
Adapter design pattern is one of the structural design pattern and it is used so that two unrelated interfaces can work together. It is often used to make existing classes work with others without modifying their source code. The pattern involves creating an adapter class that bridges the gap between the interfaces, allowing them to communicate effectively.



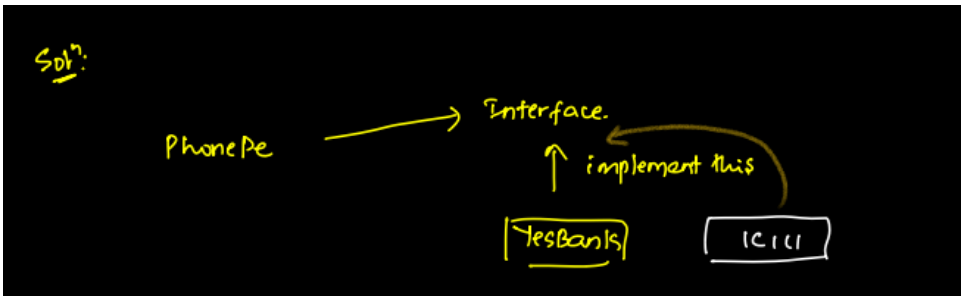
Let's build phonePe from the scratch



What are the problems in this phonePe design

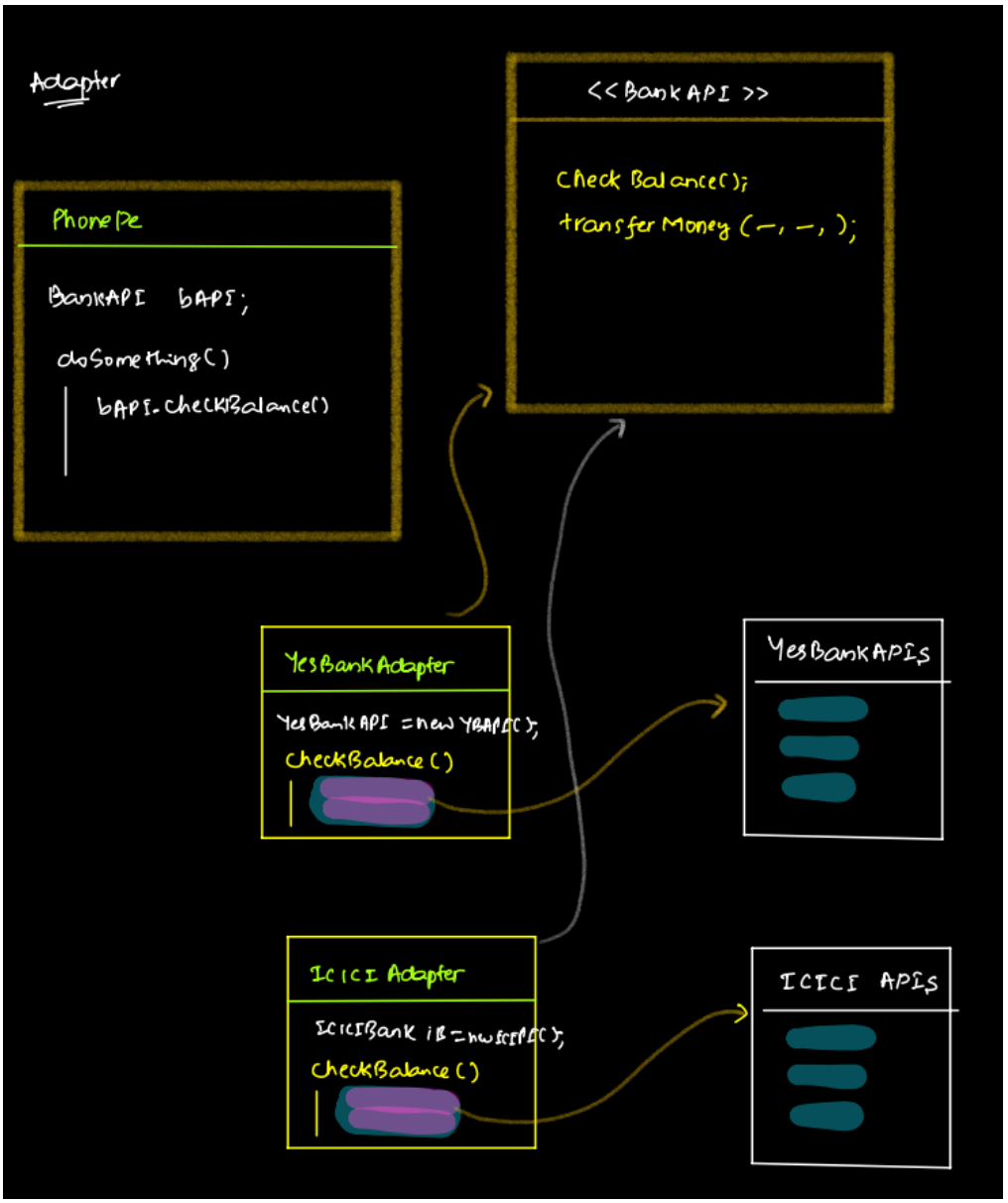


What is the solution ? Interface ? While it seems like a good idea to have a shared interface for all classes, it's not practical because other vendor classes may struggle to implement it. They might either forget to do it or provide a response that doesn't match, making this solution unreliable.

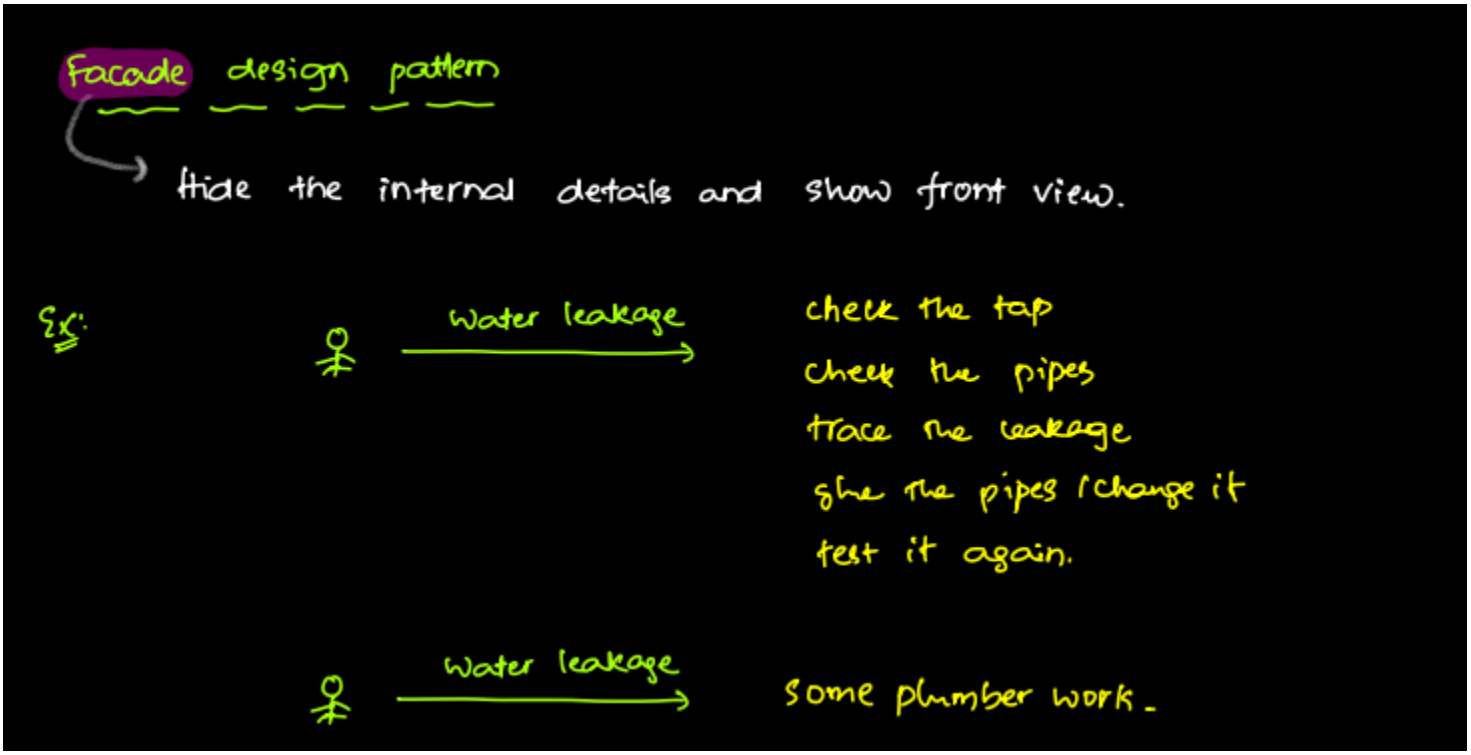


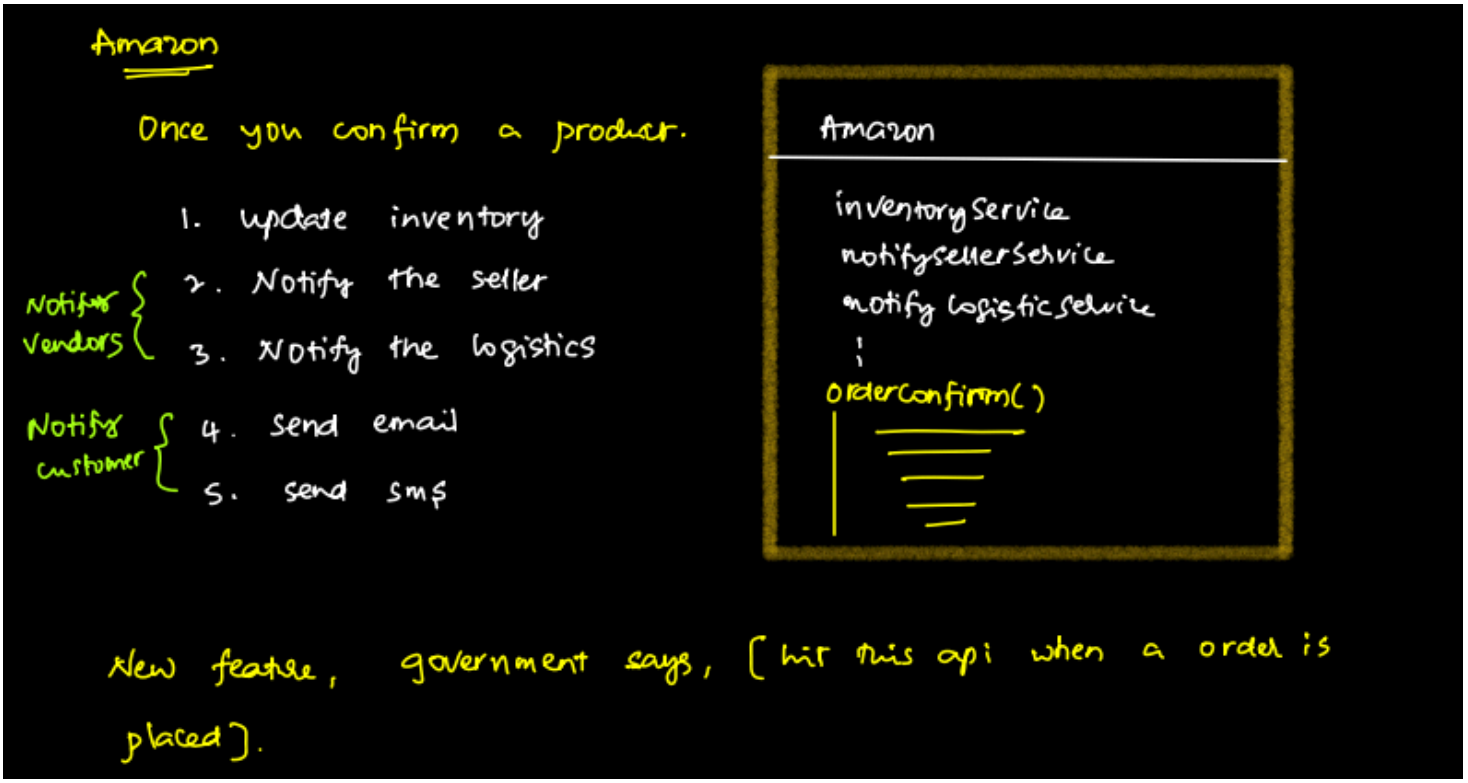
Problem with interface and how adapter solves it
https://github.com/kkumarsg/lld/tree/main/src/main/java/designpatterns/adapter

Instead of using an interface, we will address this by introducing an additional layer for all incompatible classes. This layer will handle the required methods for our class and extract data from the incompatible class.

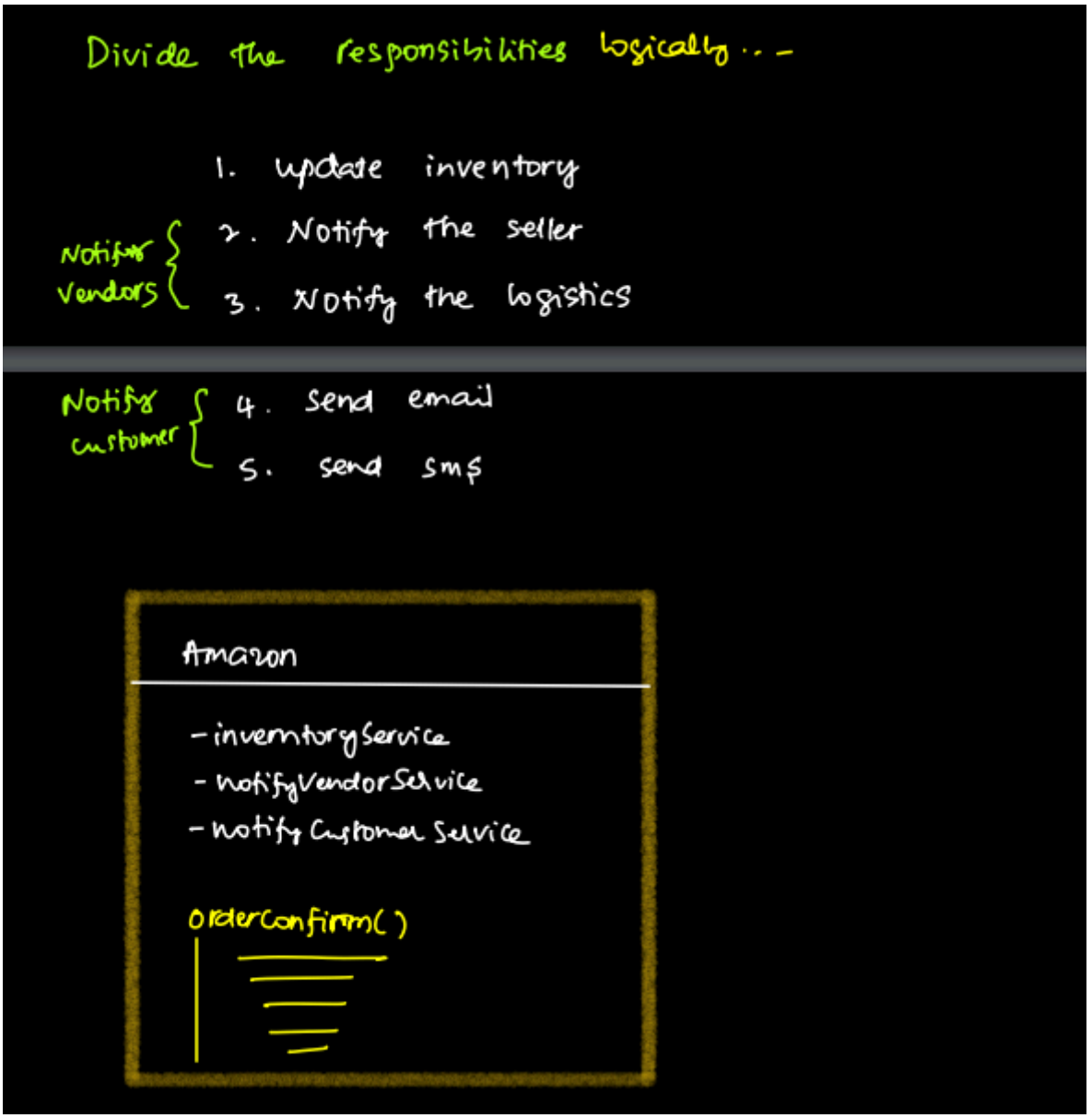


What is an Facade design pattern -> The facade pattern is typically used when. a simple interface is required to access a complex system, a system is very complex or difficult to understand, an entry point is needed to each level of layered software, or. the abstractions and implementations of a subsystem are tightly coupled.
Purpose: Provides a simplified interface to a set of interfaces in a subsystem.
It simply says **divid the responsibilities by creating helper classes.**
Example: javax.faces.context.FacesContext in JavaServer Faces (JSF) provides a simplified interface to various services like validation, conversion, and navigation





In the preceding screen, we observe a substantial "Amazon" class encompassing various responsibilities, including inventory management, vendor notifications, and customer notifications. This class incorporates numerous features and requirements, and its scope is expected to expand as new requirements arise.



To tackle this concern, we will logically partition its responsibilities and introduce additional helper classes such as NotifyVendor and NotifyCustomer. We will then instantiate these classes using the following method.

Implementation of Adapter Design Pattern -:

Suppose you are having an IPhone6s and your friend is having an IPhone4s now you went to your friend’s house but you forgot to carry your charger with you and you need to charge your phone and the charger you need is not available . Now what will you do🤷‍♀️? So here in this situation an adapter is going to help us . You will use an adapter to charge your IPhone6s from IPhone4s charger . So here you have not changed the IPhone4s charger to IPhone6s charger you just adapted the situation and have your work done . But how can we implement it in code 🤷‍♀️? Let’s see .

Step 1 -:
Define the Target interface: This interface should specify the operations that the client code expects. So here our client that is you want to charge your phone .

IPhone Interface -:
```
public interface IPhone
{
   public void OnCharge();
}
```

Step 2 -:
Implement the Adaptee class: This class represents the existing component with an incompatible interface. So this is the IPhone4s charger that your friend has .

IPhone4s Charger -:
```
public class Iphone4sCharger implements Charger
{
   Iphone4sCharger(){};
   public void charge()
   {
      System.out.println("charging with 4s charger");
   }
}
```
Charger Interface -:
```
public interface Charger
{
   public void charge();
}
```

Step 3 -:
Create the Adapter class: The Adapter class implements the Target interface and internally uses an instance of the Adaptee class. It adapts the Adaptee's interface to match the Target interface by delegating the calls appropriately. So here we will use an IPhone4s to IPhone6s adapter so that we can easily charge our IPhone6s .

IPhone4stoIPhone6s Adapter class -:
```
public class Iphone4sTo6sAdapter implements Charger
{
   Iphone4sCharger iphone4sCharger;

   Iphone4sTo6sAdapter()
   {
      iphone4sCharger = new Iphone4sCharger();
   }
```

```
    @Override
    public void charge()
    {
        iphone4sCharger.charge();
    }
}
```

So here we have created an instance of IPhone4s Charger and used its charge() method .

Step 4 -:
Connect the client code to the Adapter: Instantiate the Adapter class and use it as a bridge between the client code and the Adaptee. So here you will connect your IPhone6s to IPhone4s Charger to charge your phone .

IPhone 6s class -:

```
public class IPhone6s implements IPhone
{
    Charger Iphone4sTo6sAdapter;
    public IPhone6s(Charger iphone4sTo6sAdapter)
    {
        this.Iphone4sTo6sAdapter = iphone4sTo6sAdapter;
    };

    @Override
    public void OnCharge()
    {
        Iphone4sTo6sAdapter.charge();
    }
}
```

Main class -:

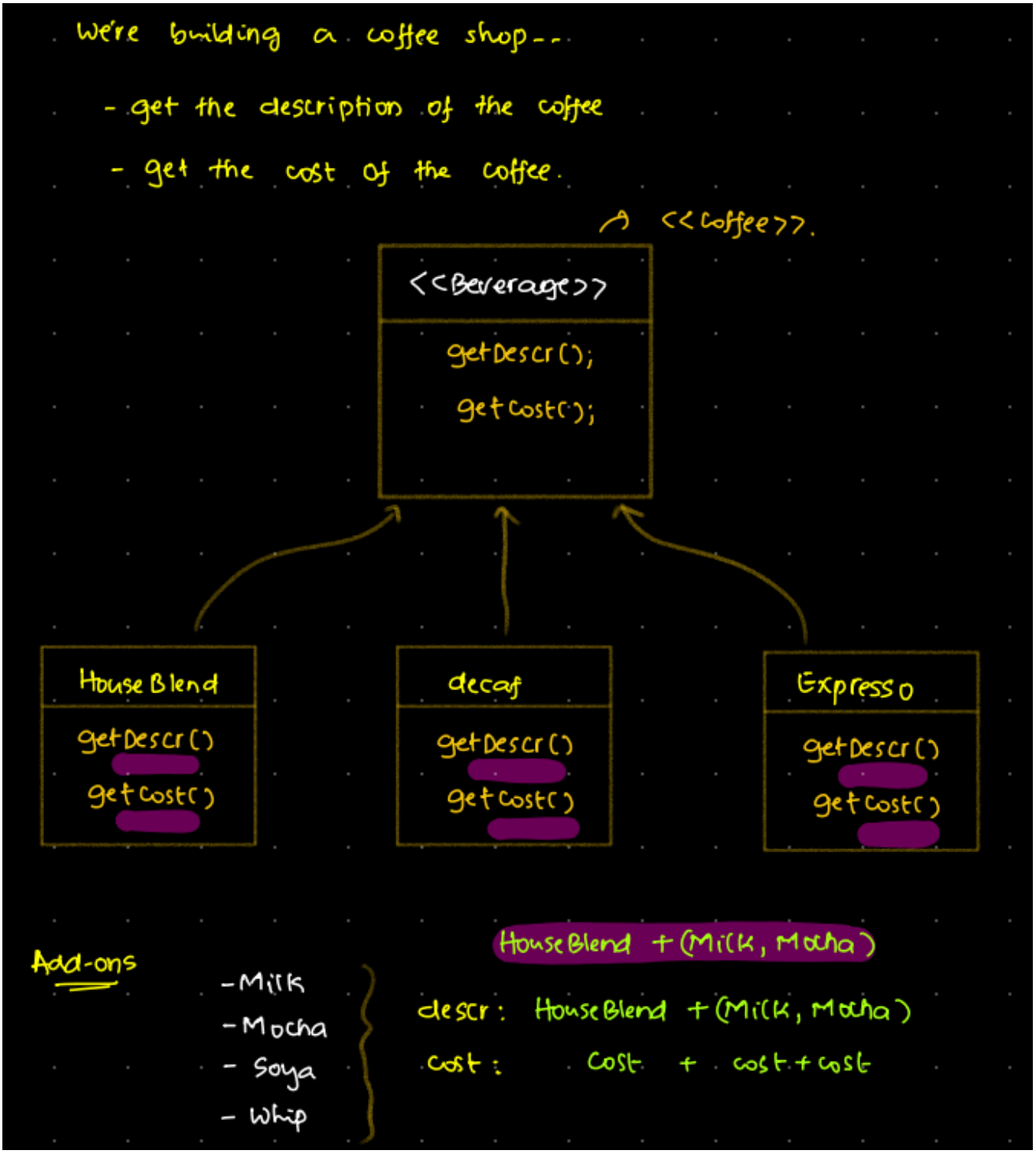
```
public class main
{
    public static void main(String args[])
    {
        IPhone6s iphone6s = new IPhone6s(new Iphone4sTo6sAdapter());
        iphone6s.OnCharge();
    }
}
```

So here you can clearly see that if you will directly use IPhone4s Charger to charge your phone it won't work but using an adapter will do so for you .

Backend LLD & Projects-2: Decorator and Flyweight Design Patter - Jan 27

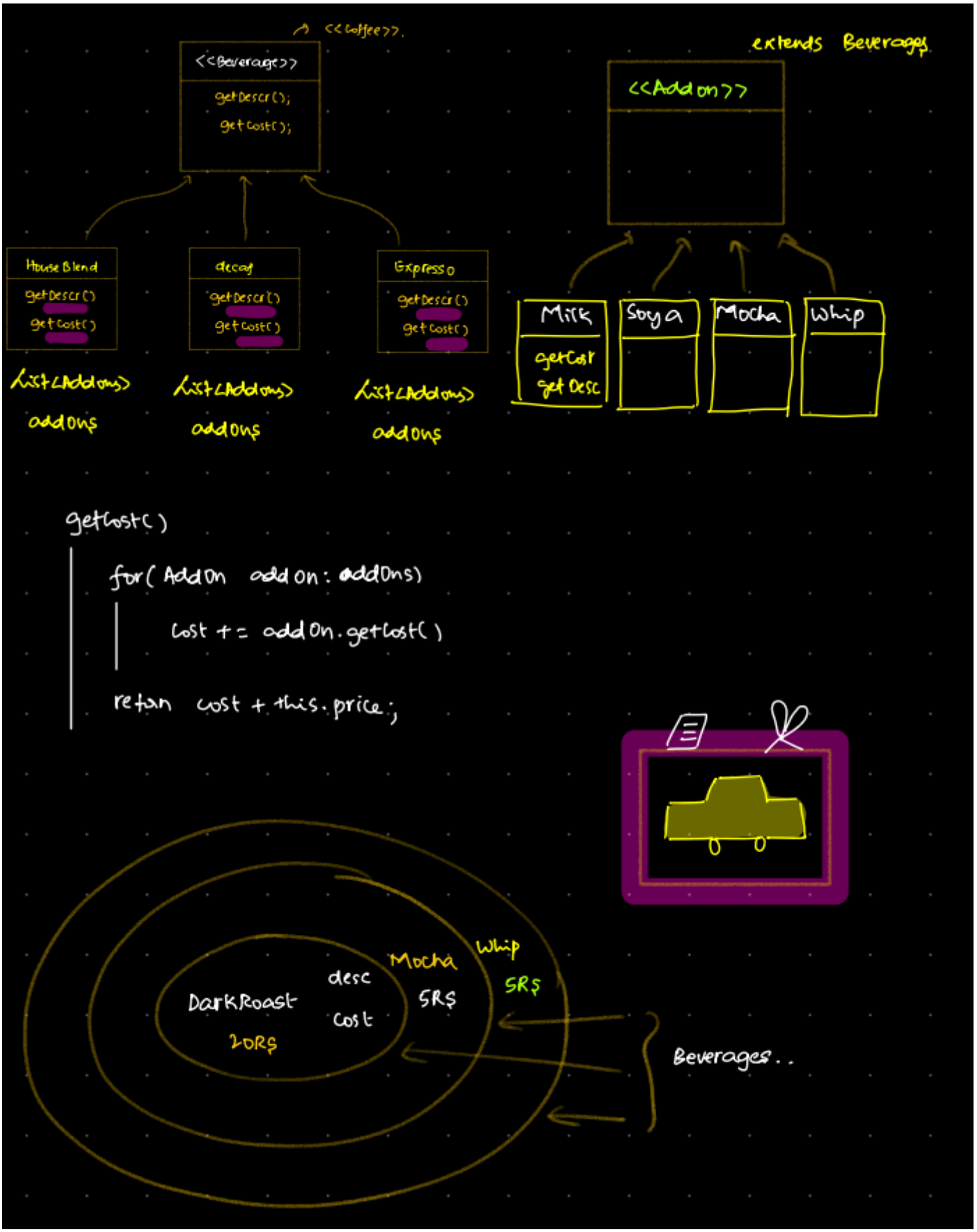
- **Decorator Design Pattern**
  - **Building a coffee shop**
  - **Analogy for decorator design pattern**
  - **Decorator design pattern code**
- **Flyweight Design Pattern**
  - **Bullet class in a game of PugB**
  - **Calculate the memory required to load 1 lakh bullets**
  - **Intrinsic and Extrinsic properties**
  - **Definition of flyweight design pattern**

Building a coffe shop ->



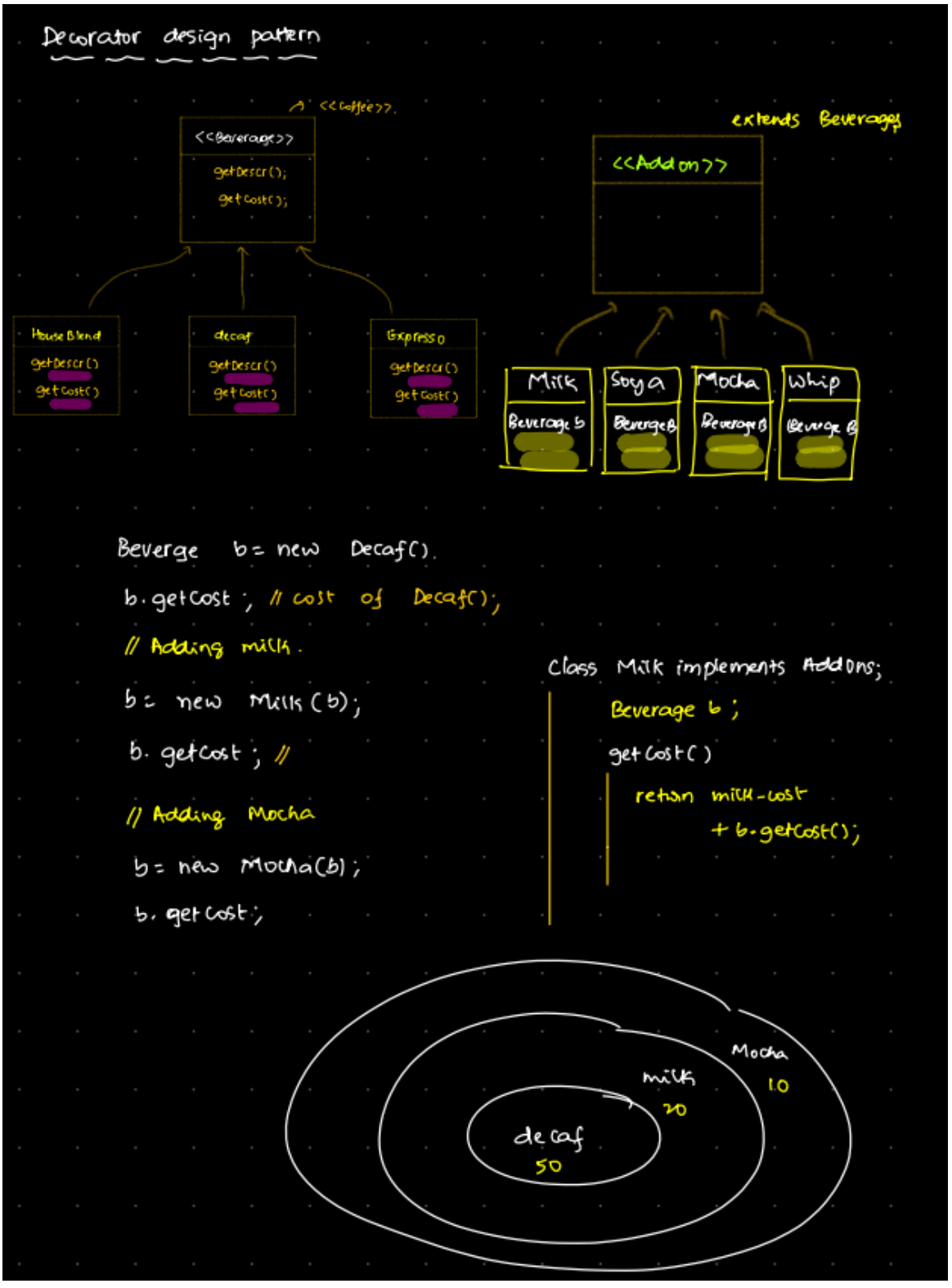
first design : each type of coffee implements beverage

As observed, we can employ the approach below where we add all the addons in the getCost method using a for loop. However, the issue with this approach is that we can only add addons at compile time, and it's not feasible to do so during runtime.



To address the dynamic addons problem, we'll use the **Decorator design pattern**. Here, we create an interface called Beverage and implement it in both the Coffee and Addons classes. We implement methods like getDes (description) and getCost. Now, we create an instance of Beverage in the addons classes and assign it through the constructor by passing a reference.

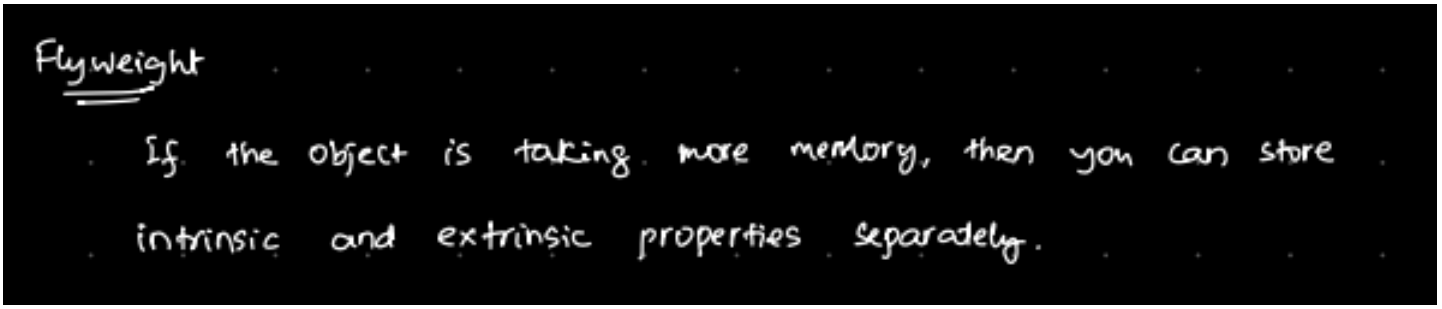
For instance, in the main class, we create a Decaf coffee object of type Beverage and call its getCost method. When adding the milk addon, we pass this Decaf instance to the milk addon. The milk addon then stores this reference and adds the cost of milk, returning the new price. The same process is followed for other addons as well. In simpler terms, we're adding multiple layers of addons to the base product, and this concept is known as decorators.



```java
package decorator;

interface Dress {
  public void assemble();
}

class BasicDress implements Dress {
  @Override
  public void assemble() {
    System.out.println("Basic Dress Features");
  }
}

class DressDecorator implements Dress {
  protected Dress dress;

  public DressDecorator(Dress c) {
    this.dress = c;
  }

  @Override
  public void assemble() {
    this.dress.assemble();
  }
}

class CasualDress extends DressDecorator {
  public CasualDress(Dress c) {
    super(c);
  }

  @Override
  public void assemble() {
    super.assemble();
    System.out.println("Adding Casual Dress Features");
  }
}



class SportyDress extends DressDecorator {
  public SportyDress(Dress c) {
    super(c);
  }



  @Override
  public void assemble() {
    super.assemble();
    System.out.println("Adding Sporty Dress Features");
  }
}
```

```
class FancyDress extends DressDecorator {
  public FancyDress(Dress c) {
    super(c);
  }

  @Override
  public void assemble() {
    super.assemble();
    System.out.println("Adding Fancy Dress Features");
  }
}

public class DecoratorPatterTest {

  public static void main(String[] args) {

    Dress sportyDress = new SportyDress(new BasicDress());
    sportyDress.assemble();
    System.out.println();

    Dress fancyDress = new FancyDress(new BasicDress());
    fancyDress.assemble();
    System.out.println();

    Dress casualDress = new CasualDress(new BasicDress());
    casualDress.assemble();
    System.out.println();


    Dress sportyFancyDress = new SportyDress(new FancyDress(new  BasicDress()));
    sportyFancyDress.assemble();
    System.out.println();

    Dress casualFancyDress = new CasualDress(new FancyDress(new BasicDress()));
    casualFancyDress.assemble();

  }
}
```

Flyweight Design Pattern -> The Flyweight Design Pattern is a structural design pattern that focuses on minimizing memory usage or computational expenses by sharing as much as possible with related objects; it's particularly useful when dealing with a large number of similar objects. The pattern achieves this by dividing an object's state into intrinsic (shared) and extrinsic (unique) parts.



Let's simplify the Flyweight Design Pattern using a metaphor

Imagine you're running a coffee shop, and you have a variety of coffee flavors. Some customers order the same flavor multiple times. Instead of creating a new coffee cup for each order of the same flavor, you decide to reuse the cups you already have.

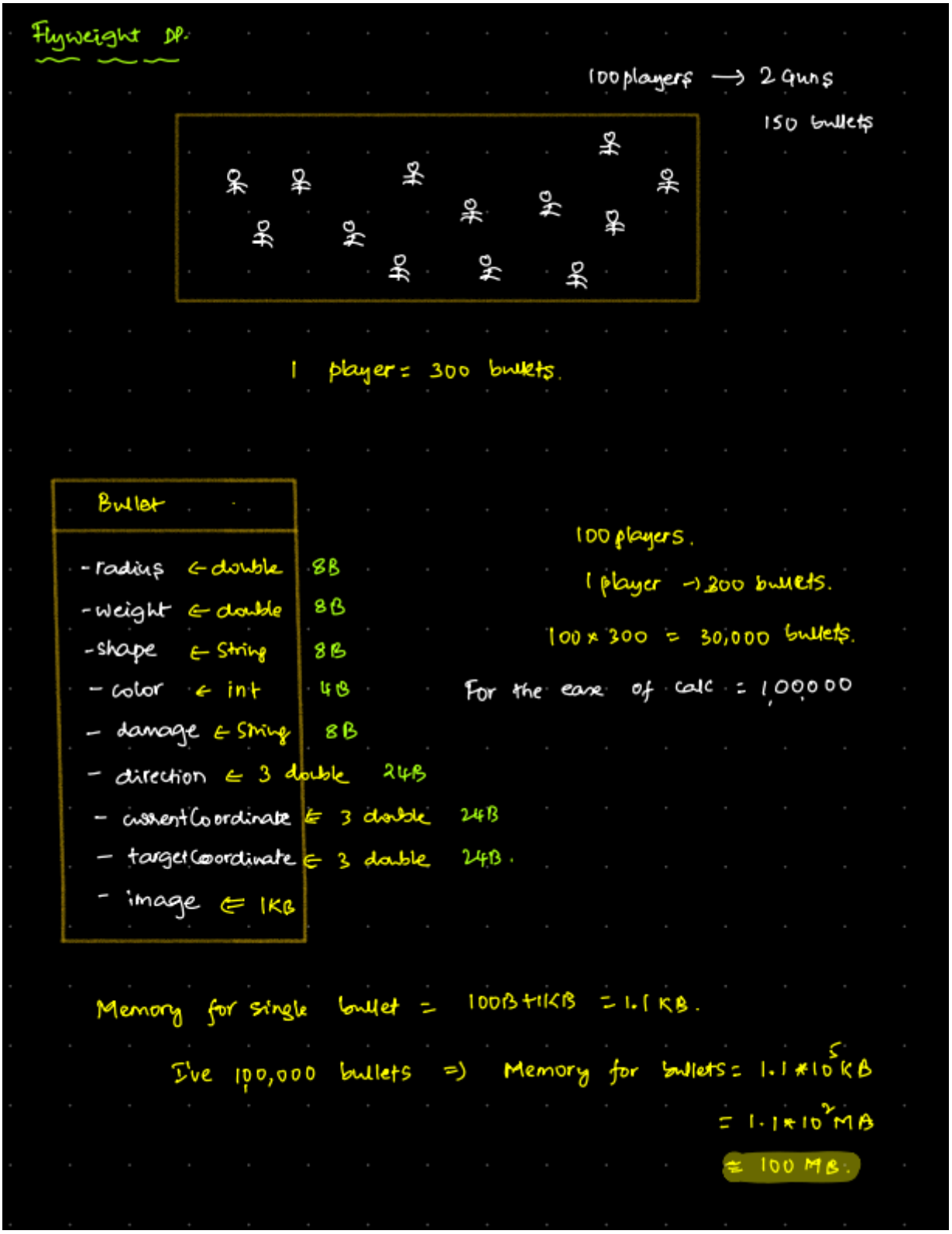
In this metaphor:
CoffeeFlavor: This is like a cup that holds a specific flavor. You don't create a new cup for each order; you reuse existing cups.

CoffeeFlavorFactory: This is like your storage room where you keep all the different cups. When a customer orders a flavor, you check if you already have a cup for that flavor. If you do, you reuse it; if not, you create a new cup and store it for future use.

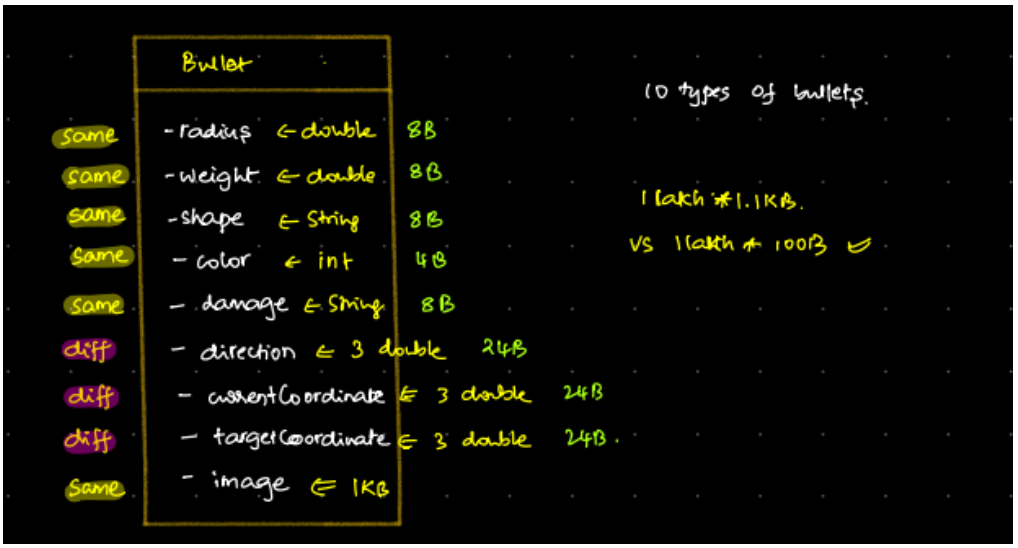
CoffeeOrder: This represents a customer's order, and the context (extrinsic state) is the table number. The coffee cup (flavor) is shared among orders to save resources.

So, the Flyweight pattern helps you save resources (cups in this case) by reusing shared components when possible, making your coffee shop more efficient.

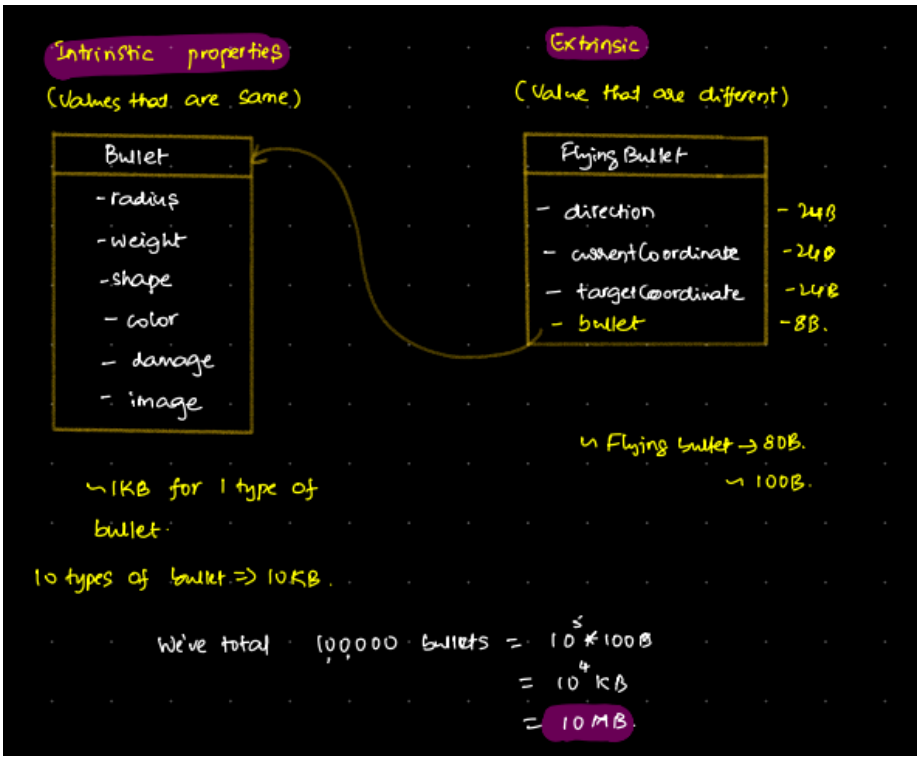
In the example below, we use the game PUBG as a reference. Around 100 players are playing the game, and we are determining the memory used by a single bullet. We then extrapolate this calculation to account for 300 bullets per player and further to estimate the total space required if we create a separate instance for every bullet. This method results in significant memory consumption, prompting us to explore ways to minimize this usage.



Calculate the memory required to load 1 lakh bullets



Intensive Property -> This is the property which is common for all objects
Extensive Property -> This is the property is unique for objects



Definition of flyweight design pattern -> To mitigate memory consumption concerns, we categorized properties into Intensive and Extensive types. We designed a single class that utilizes a constructor to store Intensive properties, and a method that accepts Extensive properties and applies them to the stored object. Additionally, we implemented a registry for caching, checking if an object has already been created. If the object is absent, we add it to the registry; otherwise, we return the existing instance. By employing a map within the class to store object types in the registry, we ensure that there is consistently only one instance of an object, even when multiple instances of the extensive class are generated. This optimization significantly enhances memory efficiency.

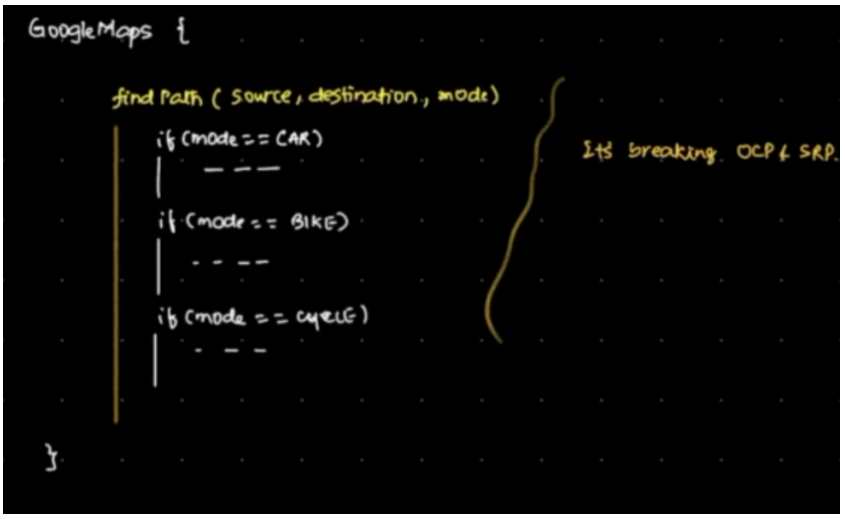
Backend LLD & Projects-2: Behavioural Design Pattern - Jan 28

- **Behavioural design pattern**
  - **Strategy design pattern**
    - **Will making separate methods help to resolve SRP issue?**
    - **Using interfaces effectively, lead to Path Calculator**
    - **Using strategy with factory**
- **Observer design pattern**
  - **Code modified using observer design pattern**

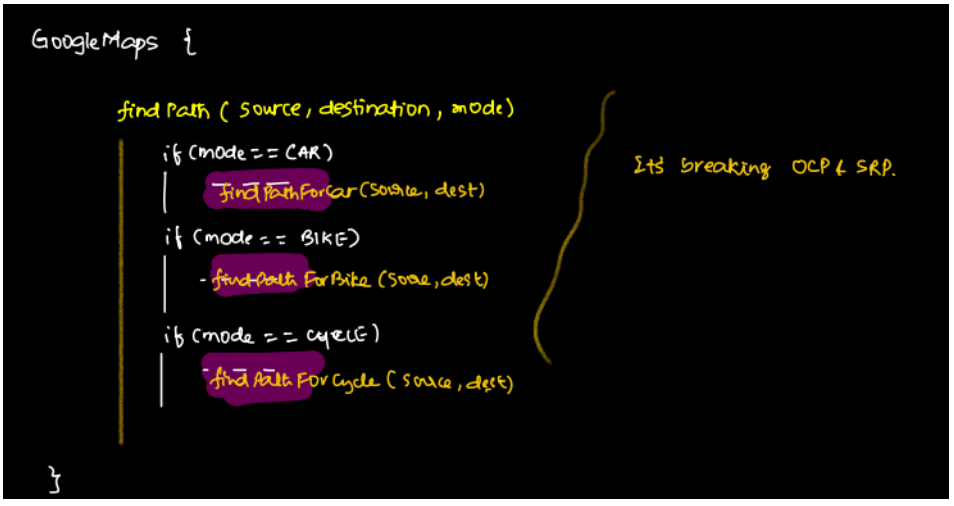
Behavioral design patterns -> focus on how objects collaborate and communicate with each other. These patterns define the patterns of communication between classes and objects to enhance flexibility and maintainability. Some common behavioral design patterns in Java include:

- **Strategy Pattern**
- **Observer Pattern**
- Command Pattern

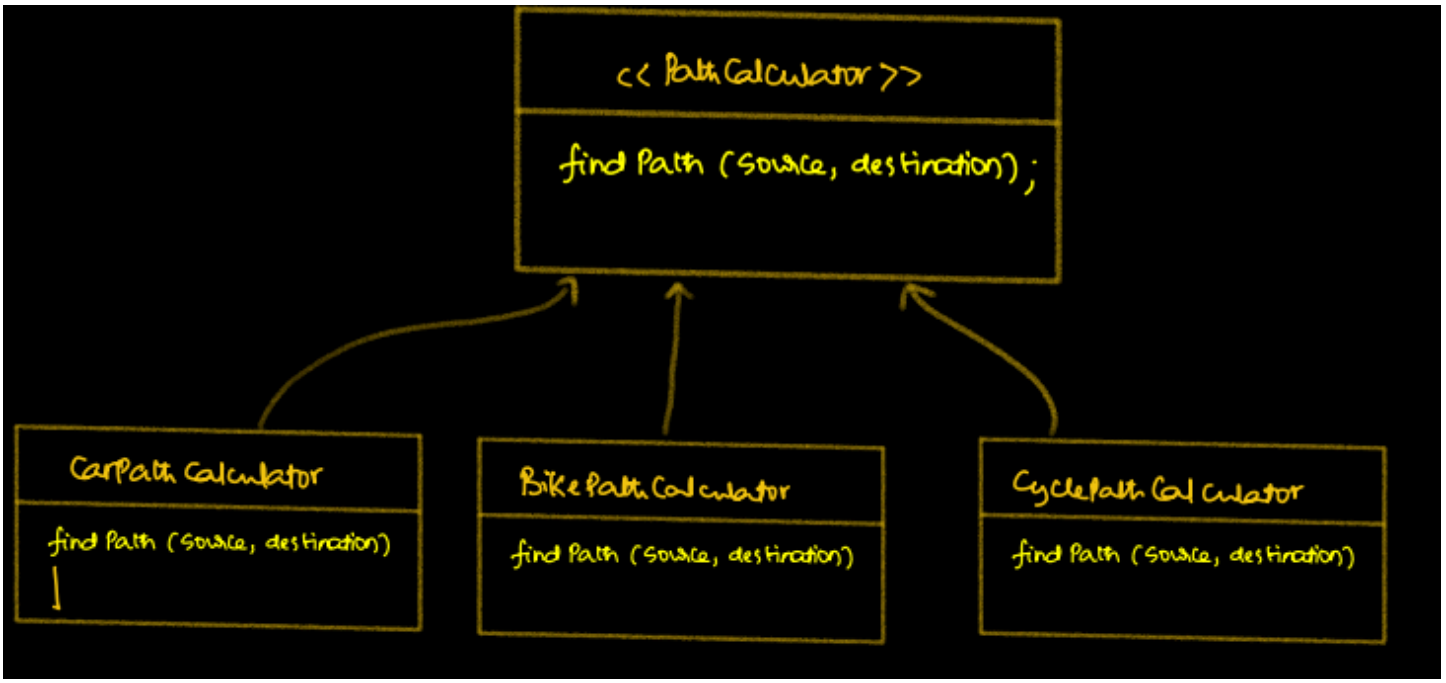
As we can see in below code we are breaking SRP and OCP solid rules using conditional rendering the code.



Will making separate methods help to resolve SRP issue? No because still we have to come and change the code if have to add one more method or change any method.



Using interfaces effectively, lead to Path Calculator -> We can address this by employing an interface. A common interface will be created, featuring a method named findPath. Subsequently, classes for all scenarios will be developed, each implementing this interface. In the client class, an instance of PathCalculator will be instantiated.

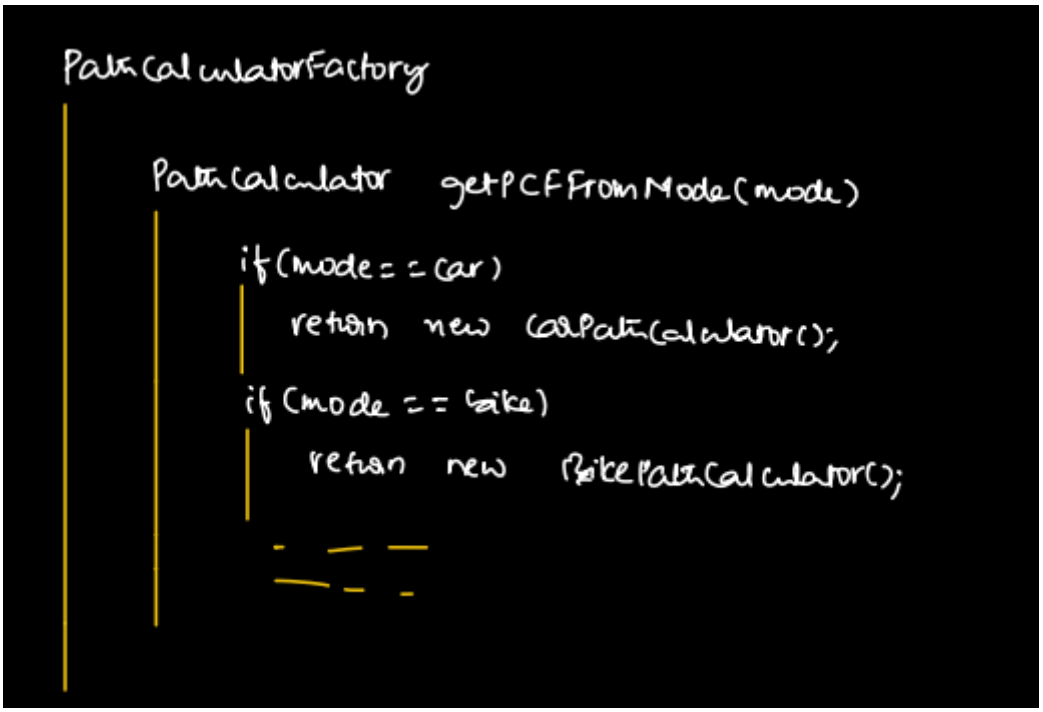



```
GoogleMaps {

        PathCalculator pathCalculatorStrategy;

        findPath ( source, dest)
        {
            pathCalculatorStrategy. findPath (source, dest);
        }

}
```

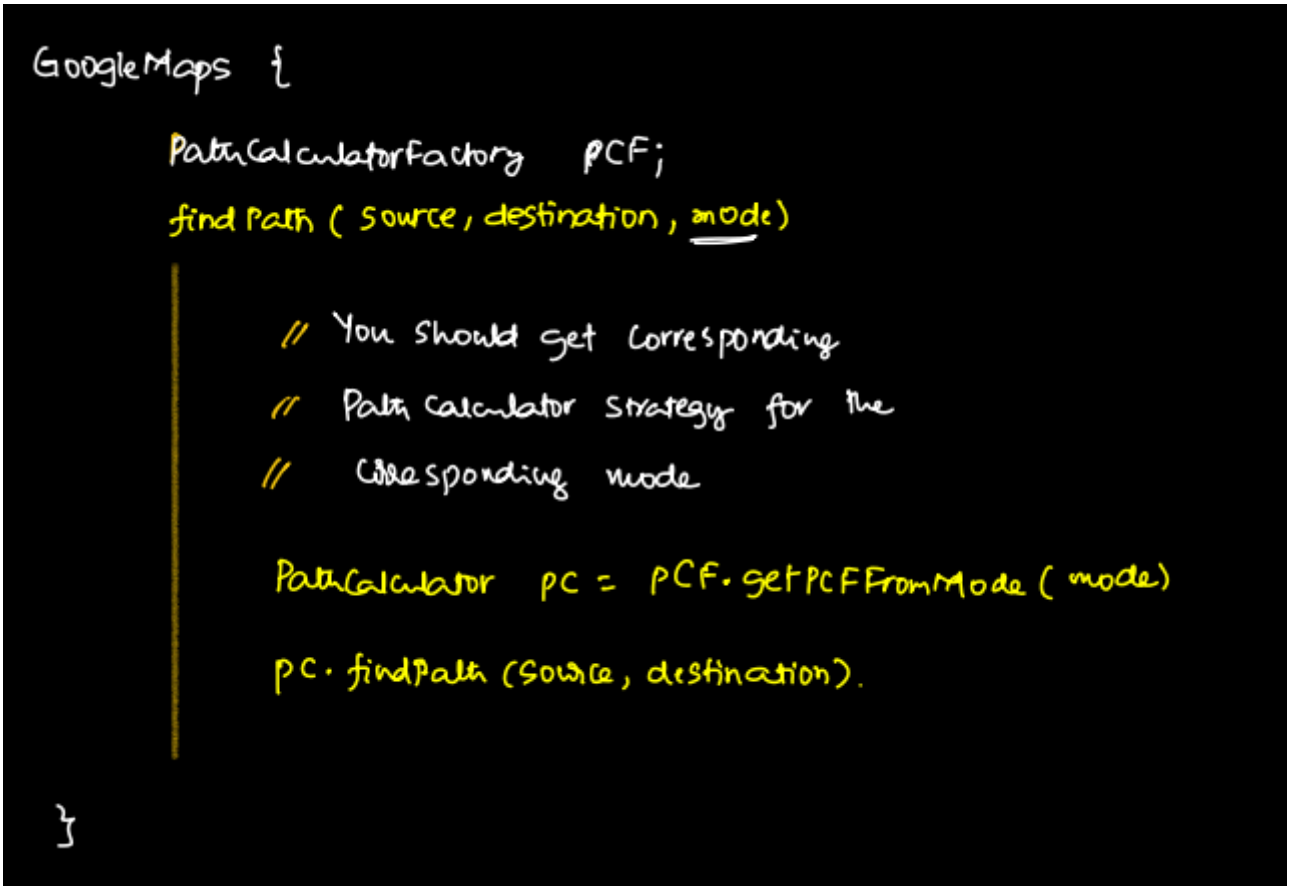
Note: Whenever we've multiple ways to do something, we can use Strategy design pattern. [You see that the ways of doing something is merely to grow].

Using strategy with factory

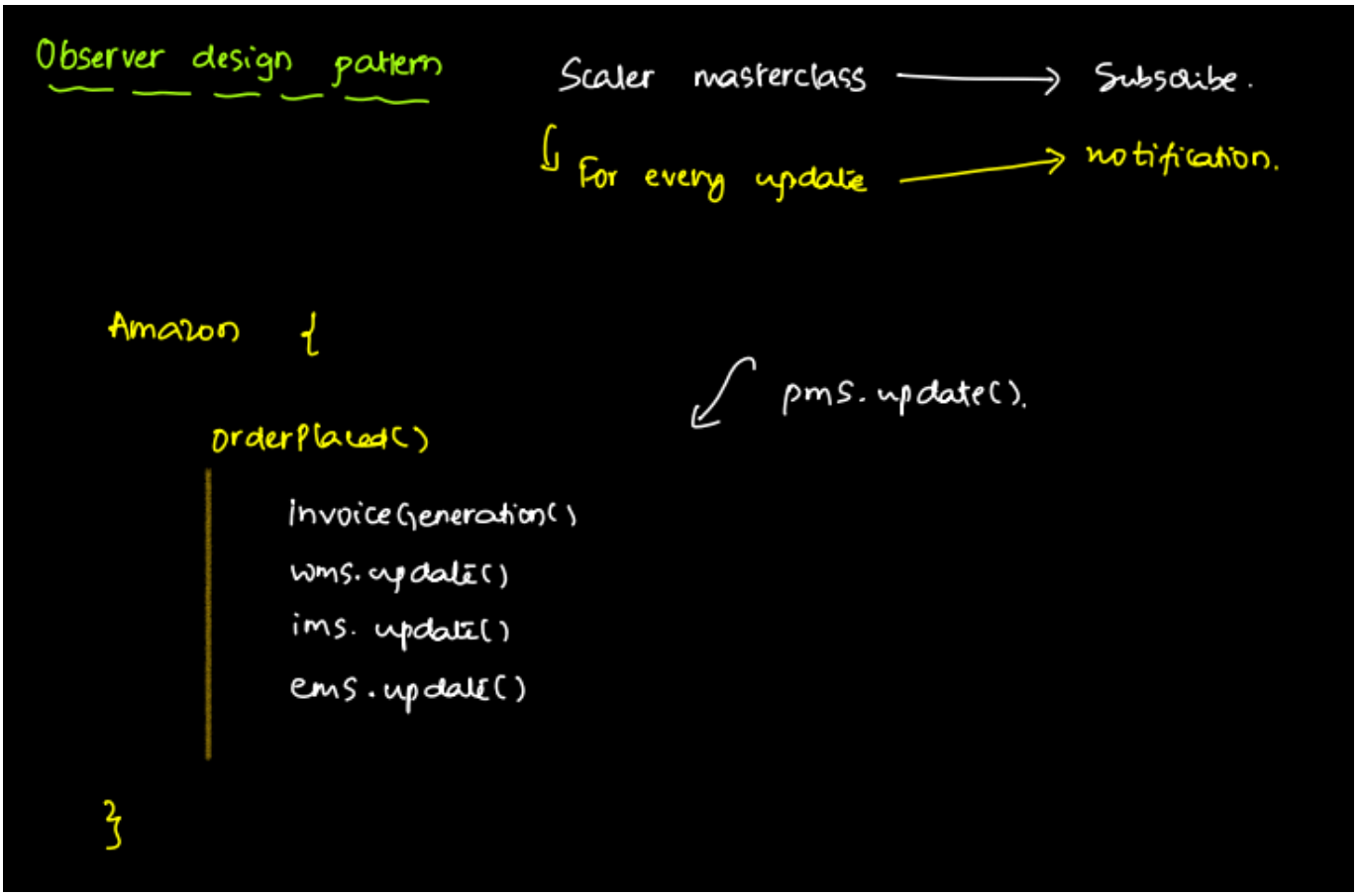
In certain cases, a class might not be permitted to directly call the findMethod. To address this concern, we can employ a Factory Method as illustrated below.



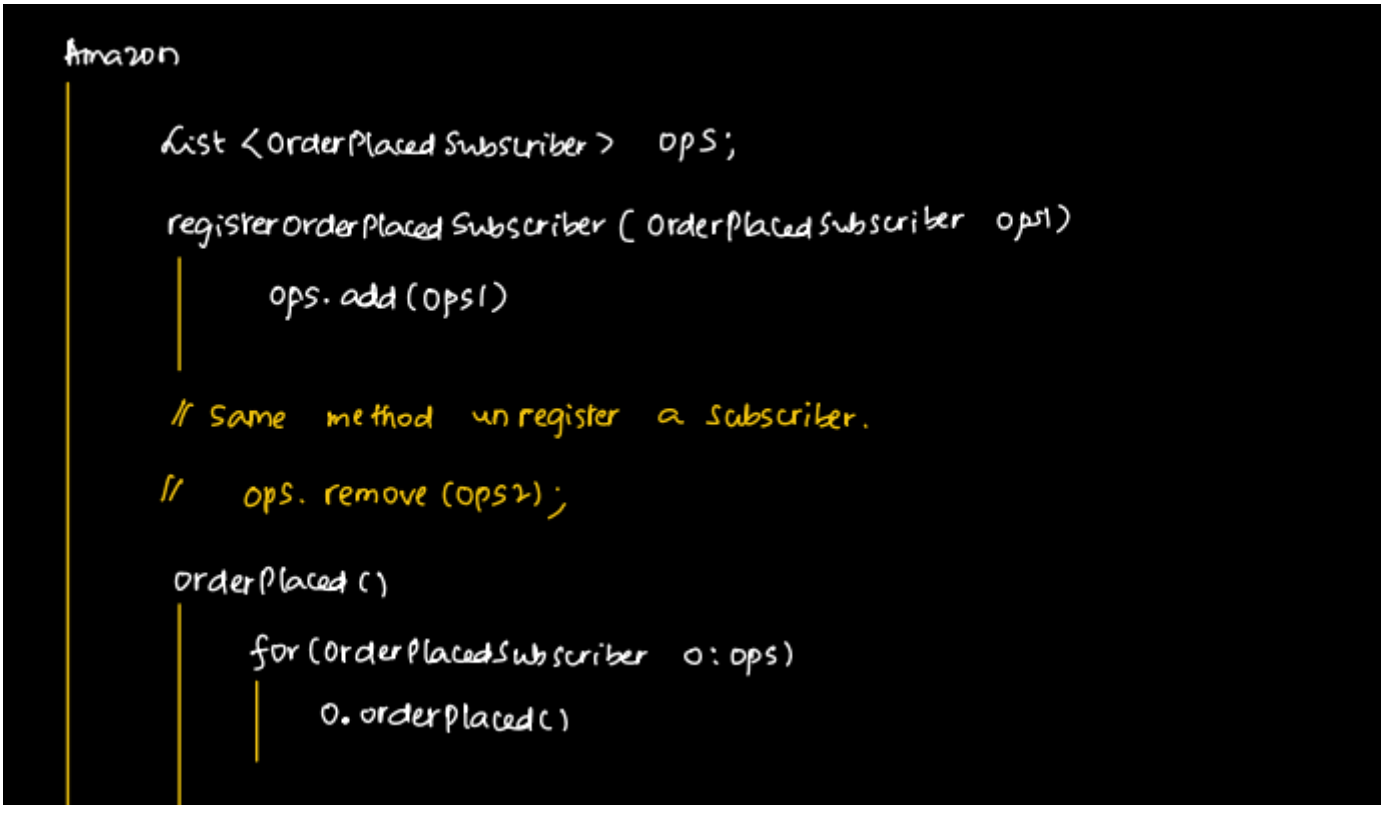
The client class will implementthe findPath method and call factory method and pass the mode type, calling the findPath method using the PathCalculator instance.



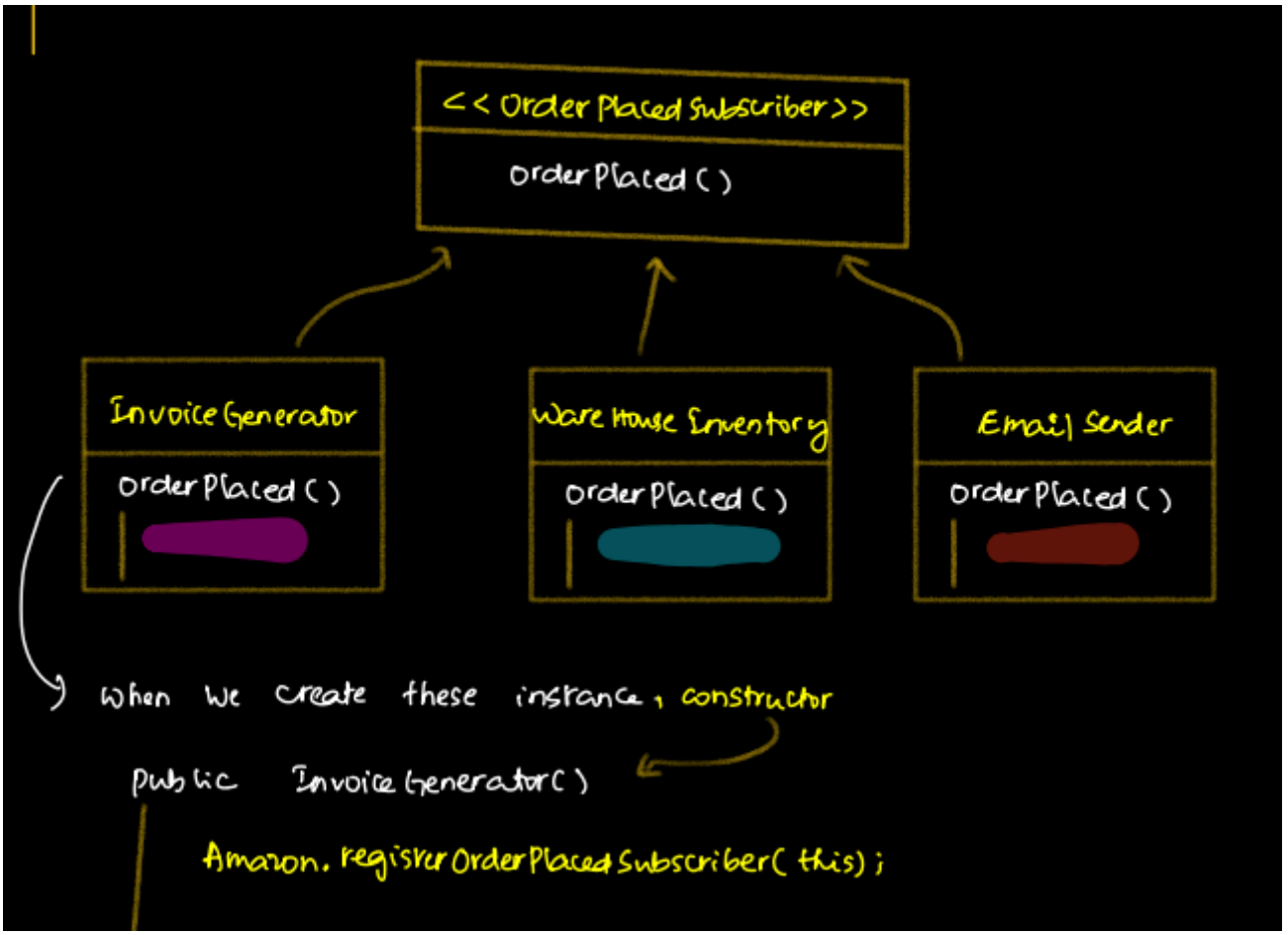
Observer design pattern ->
Defines a one-to-many dependency between objects, so that when one object changes state, all its dependents are notified and updated automatically.
Example: The implementation of event listeners in graphical user interfaces.



In the example above, all the update methods are associated with invoice generation. When creating an invoice, it is necessary to notify all these methods.



To incorporate subscription functionality, as evident in the above example, the Amazon class creates a list of objects with the type of OrderPlacedSubscriber interface. It provides methods such as register (subscribe), remove (unsubscribe), and orderPlaced. These methods iterate through the list of subscribers, invoking the orderPlaced method. Other subscribers implement this method through the common OrderPlacedSubscriber interface as below screen.

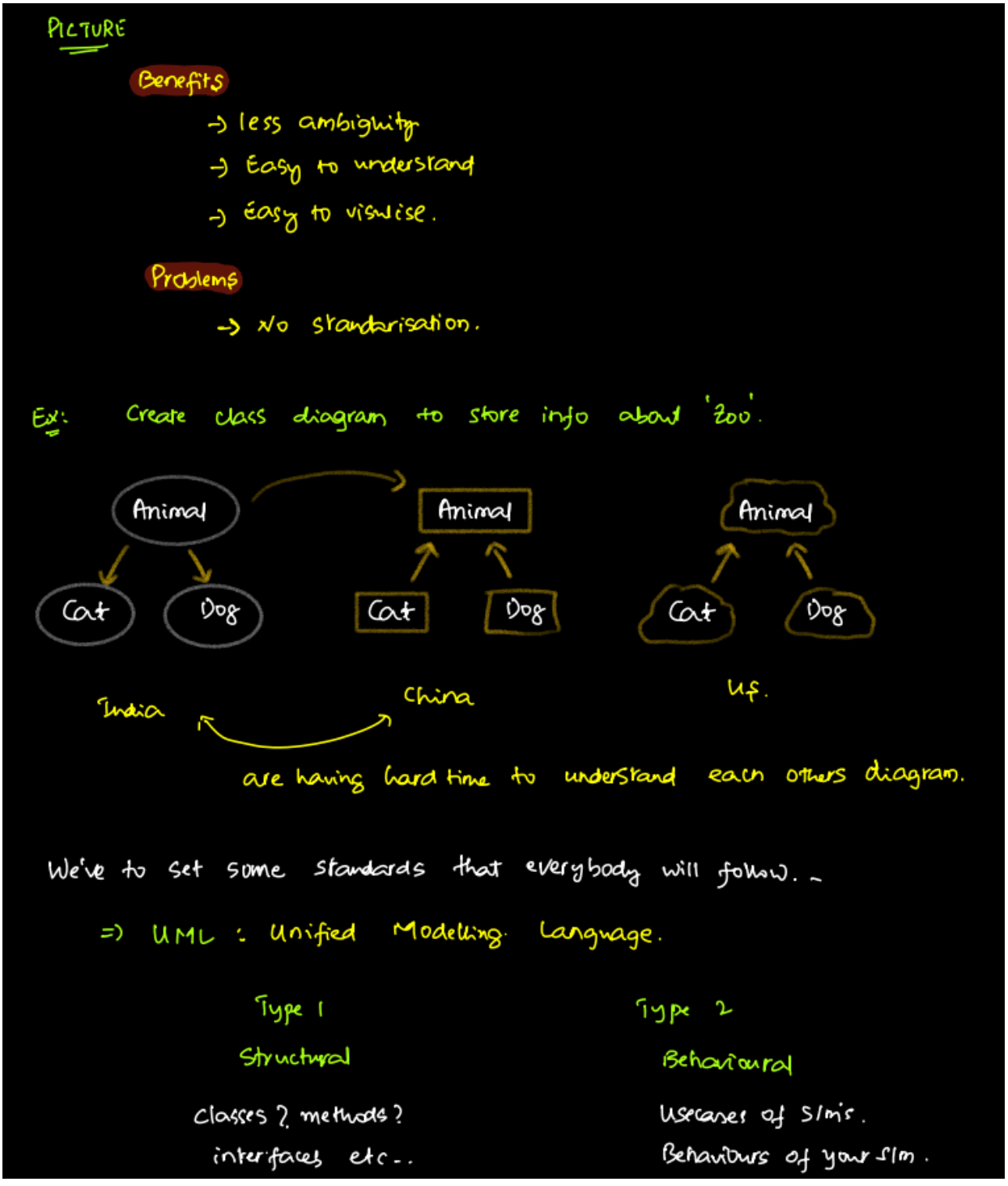


When creating an InvoiceGenerator, the process entails adhering to the interface and implementing the orderPlaced method. Following this, the subscriber is registered with the Amazon class utilizing the registerSubscriber method. The current object, stored in the list, is passed during registration. Consequently, whenever an order is placed, the subscriber is notified through the invocation of the orderPlaced method.

Backend LLD-2: UML Diagrams - Jan 30

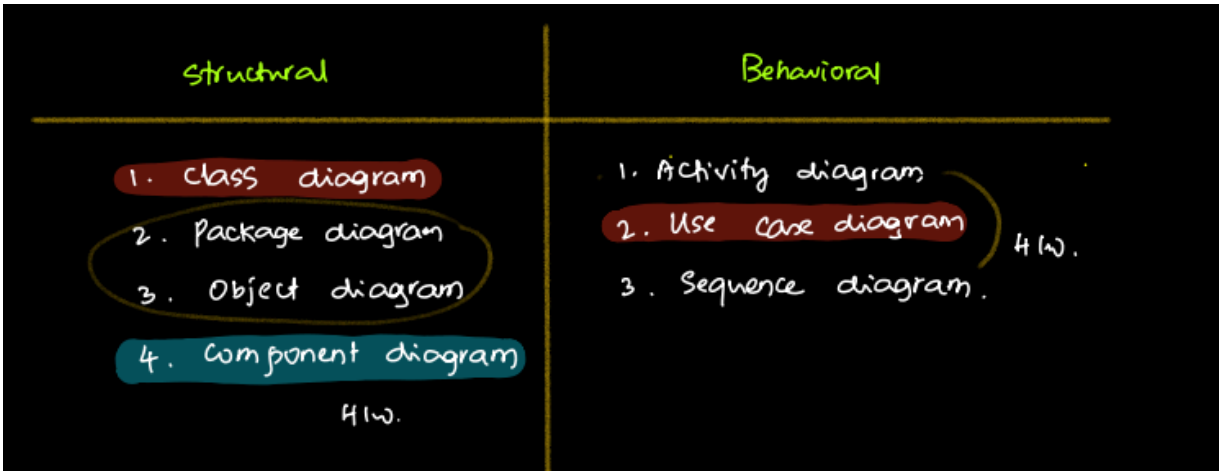
- **UML introduced**
- **Use case diagram**
- **Class diagram**

UML Introduction -> UML stands for Unified Modeling Language. It is a standardized visual modeling language design and document software systems. UML provides a set of graphical notations and diagrams that help developers and other stakeholders to understand, visualize, and communicate different aspects of a software system, such as its structure, behavior, and interactions. It serves as a common language for software architects, designers, and developers to express and share their ideas throughout the software development process.



In the example we just saw on the screen, UML (Unified Modeling Language) is needed when different developers create the same diagram with different shapes. This makes it hard to understand and standardize. UML is like a common language in the software world that helps everyone communicate and work together. There are two main types of UML diagrams:

Structural: This is about things like classes, methods, and interfaces.
Behavioral: This is about how systems behave and the different situations they can



Behavioral diagrams -> UML focus on illustrating the dynamic aspects and interactions within a system. They showcase the ways different components collaborate and respond to various events. The primary types of UML behavioral diagrams include:

Use Case Diagrams: Display how users or external systems interact with a system. They highlight the different use cases and scenarios.

Activity Diagrams: Illustrate the workflow or flow of activities within a system. They show the sequence of tasks and how they relate to each other.

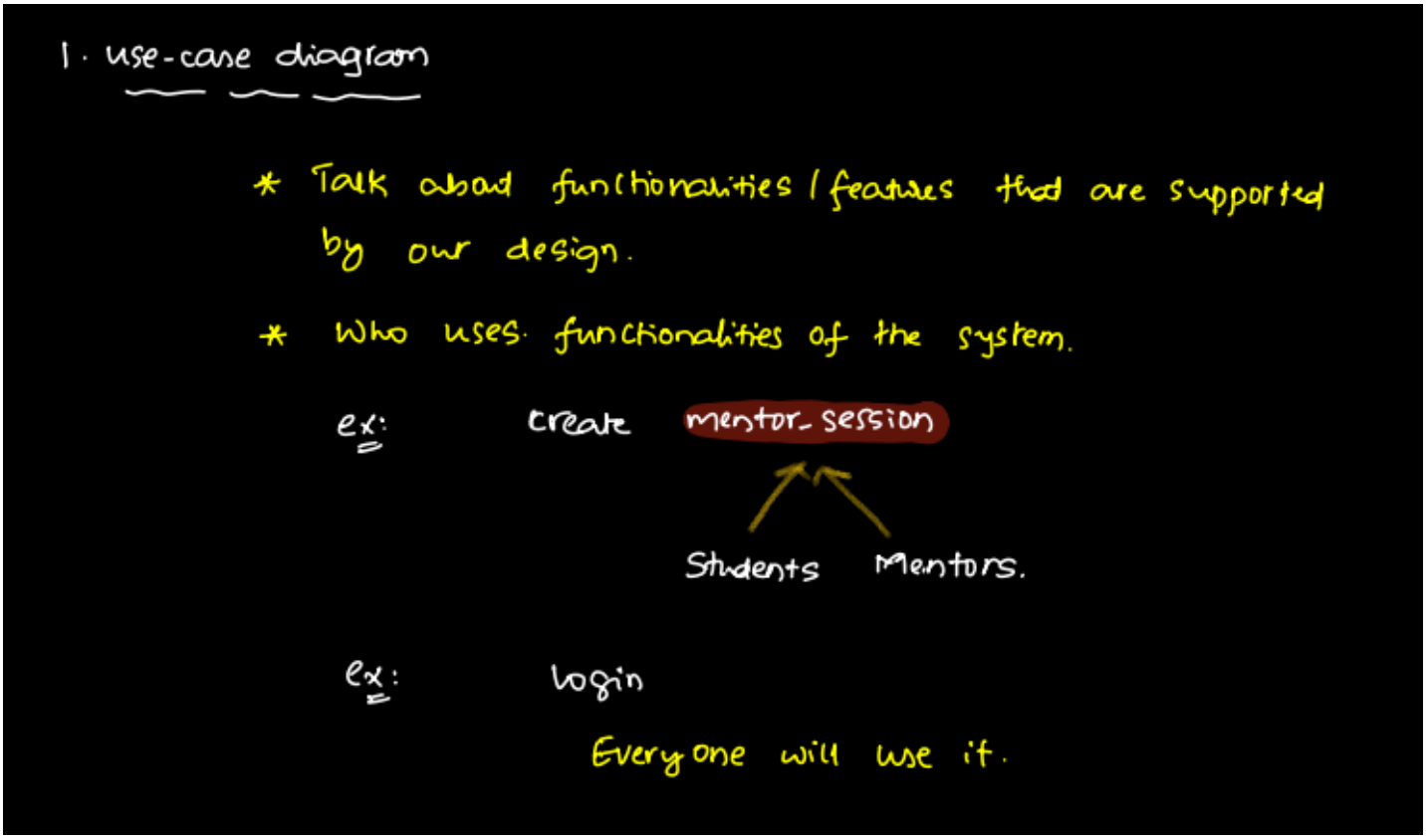
Sequence Diagrams: Visualize the interactions between different objects or components over time. They demonstrate the order of messages and the flow of control.

Collaboration Diagrams (Communication Diagrams): Similar to sequence diagrams, these depict how objects interact but focus on emphasizing the structural organization of objects and their associations.

State Machine Diagrams: Demonstrate the various states a system or object can be in and how transitions occur based on events.

These behavioral diagrams help in understanding the dynamic behavior, scenarios, and interactions that take place within a software system.

In the use-case diagram for a scaler below, we observe two functions: "create" and "login." In this diagram, we have two actors - Students and Mentors. The "create" function is accessible to both actors, while the "login" function can be utilized by all actors in the system.



There are 5 keyword to create UML diagram

System Boundary: Think of this as the invisible line that defines the scope of your system. It helps you identify what's inside your system and what's outside.

Use-case: A use-case is like a story that describes how a person or something outside the system interacts with it. It highlights specific functionalities or actions the system can perform.

Actor: An actor is like a character in your use-case story. It represents a person, a device, or another system that interacts with your system. Actors are outside the system and play a role in your use cases. Here we will use Stick diagram like draw all actor in one like

Includes: This is a way to show that one use-case can include another. It's like saying, "If you do this, it also involves doing that other thing". like Payment is a necessary step for completing the checkout process. In this context, the checkout depends on the successful occurrence of the payment; if payment doesn't happen, the checkout process will remain incomplete.

Extends: This is used to indicate that a use-case can be extended with additional functionalities under certain conditions. It's like saying, "If something special happens, we can add more to the story."

How to create use-case diagrams?

[5 key words]

① System Boundary

Every feature that is supported by my system will be inside the boundary.

← scope of my sim.

Payment conf is handled by 3rd party.

=) not part of sim boundary.

② Use-case [it's just a feature | what can you do?]

[verb].

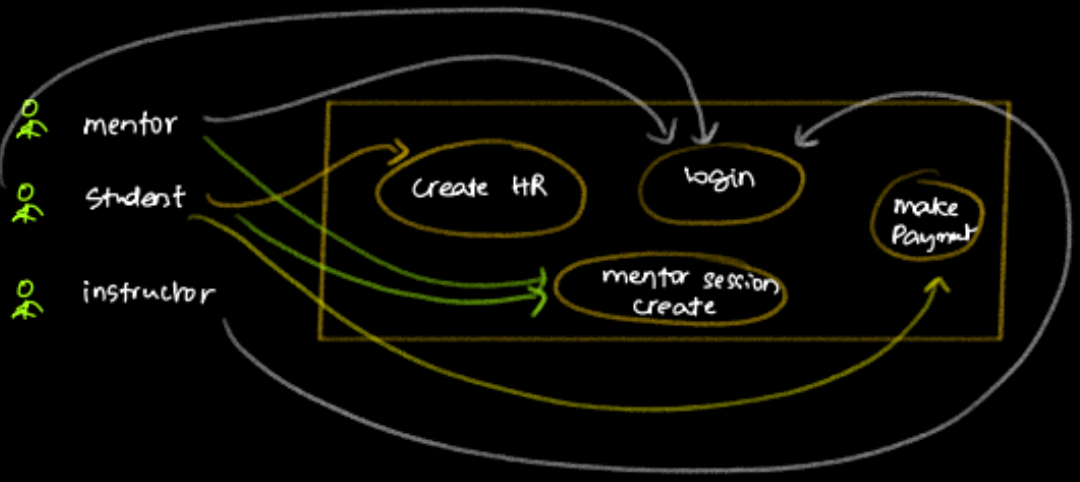
→ functionalities / features / actions

→ Represented by an oval.

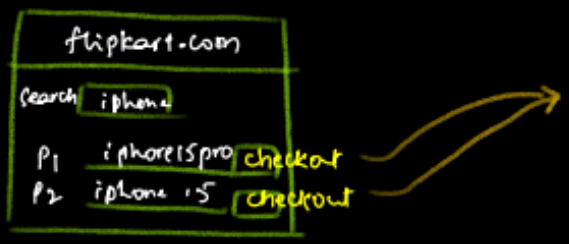
ex:

3. Actor

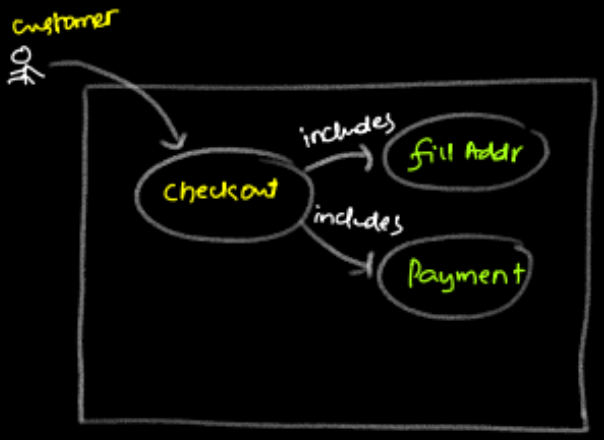
1. Users of a particular use case [Noun]
2. Stick diagram repr
3. Draw an arrow indicating rel^n b/w usecases & an actor.



4. Includes

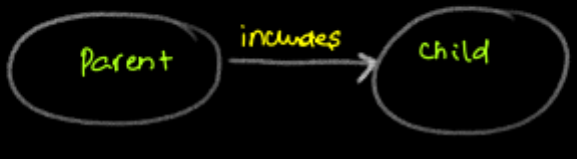


Usecase.
→ Select addr / new addr
→ Make Payment



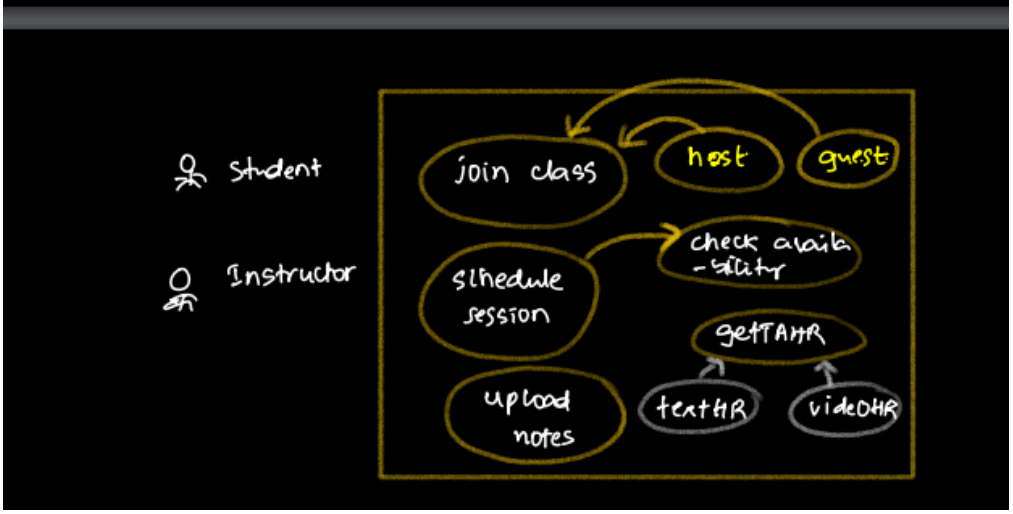

```
Void checkout()
    fillAddress()
    makePayment()
```

To complete checkout you need fill Addr & make Payment.
[dependency].

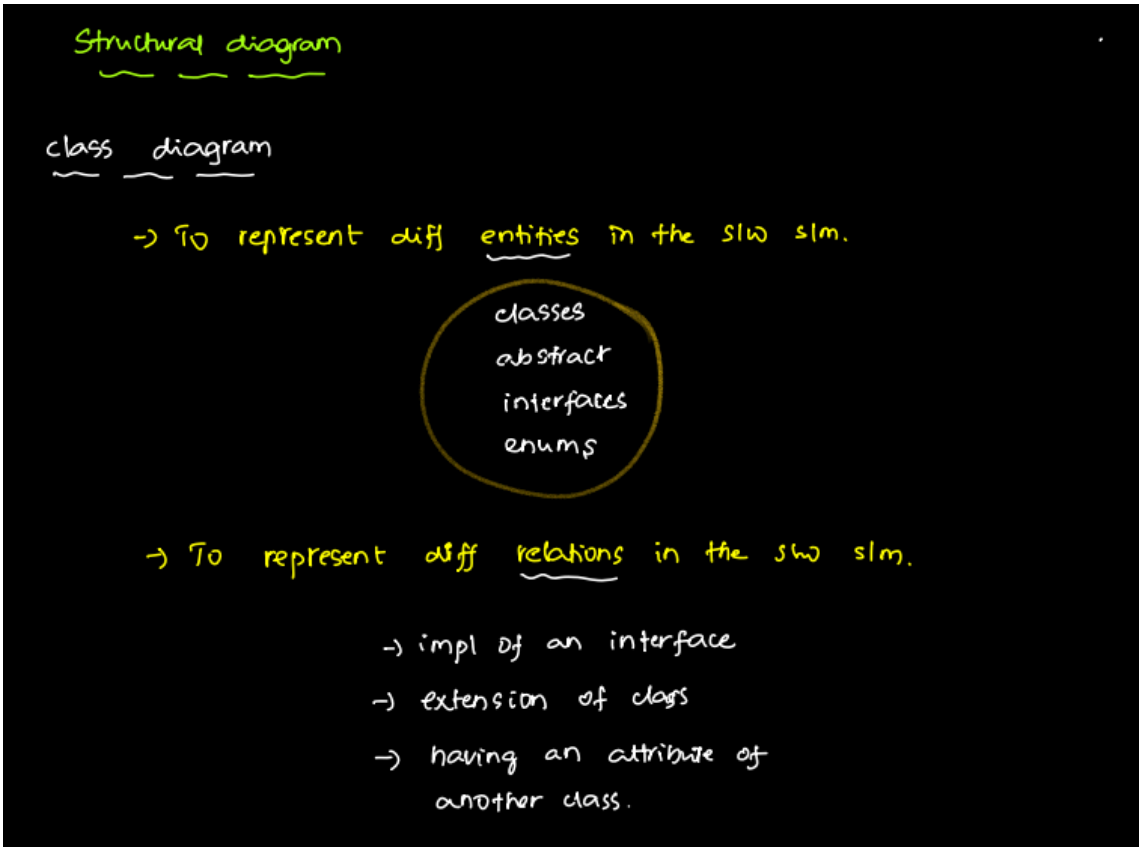


Q: Use case diagram for Scaler platform

1. 5 use cases
2. 2 Actors
3. 1 use case includes another use case
4. 1 use case which should've special use case
   // something other than login.



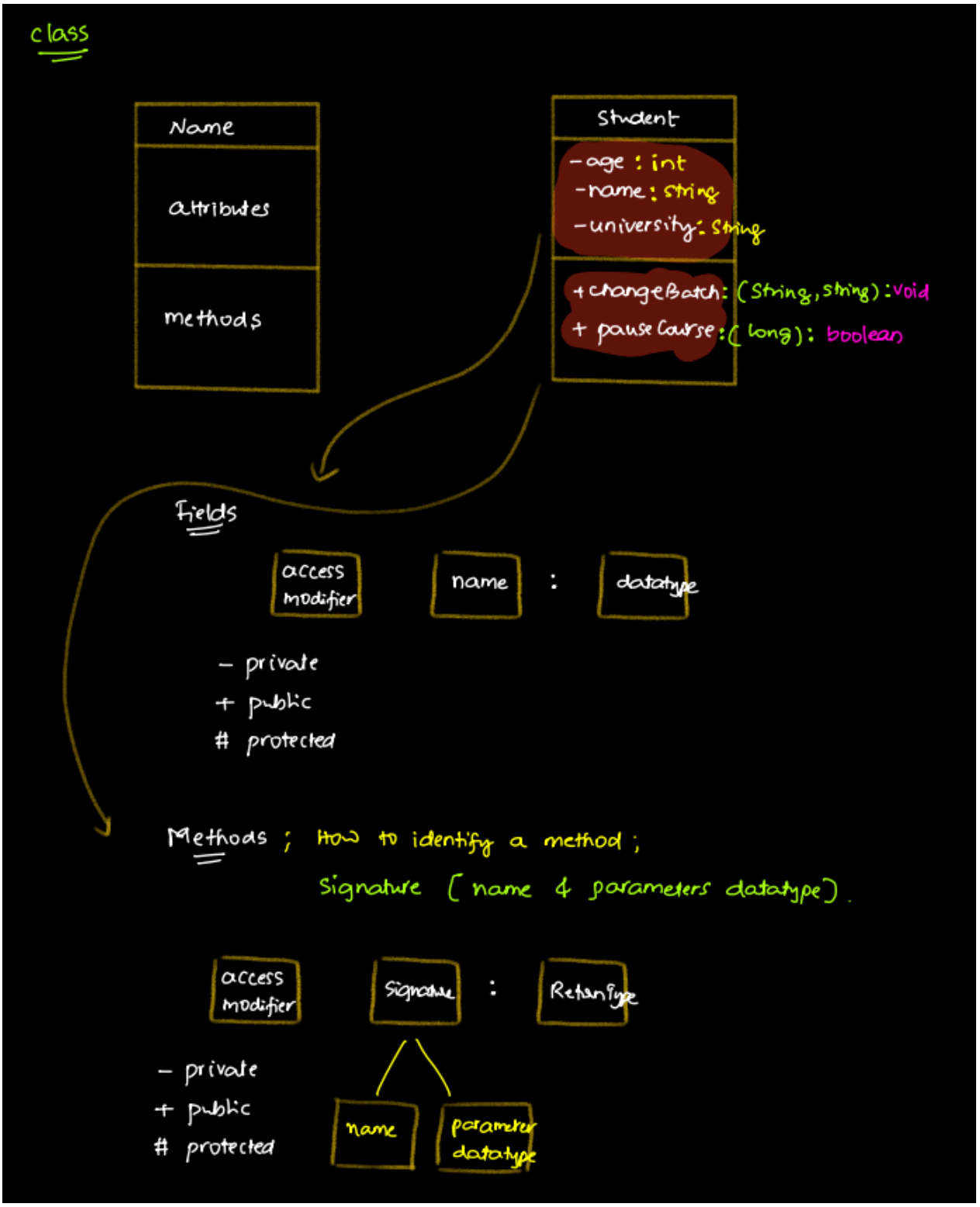
Structural diagram -> Its provides a visual representation of the static structure or composition of a system. It focuses on illustrating the different components that make up a system and how they are organized. The key elements in a structural diagram include:



Class Diagram: This diagram represents the classes in a system, along with their attributes and relationships. It's like a blueprint that shows the structure of the classes and how they interact. It will be described using a 3-row box. The first row contains the class name, the second row lists the class attributes, and the third row outlines the class methods.

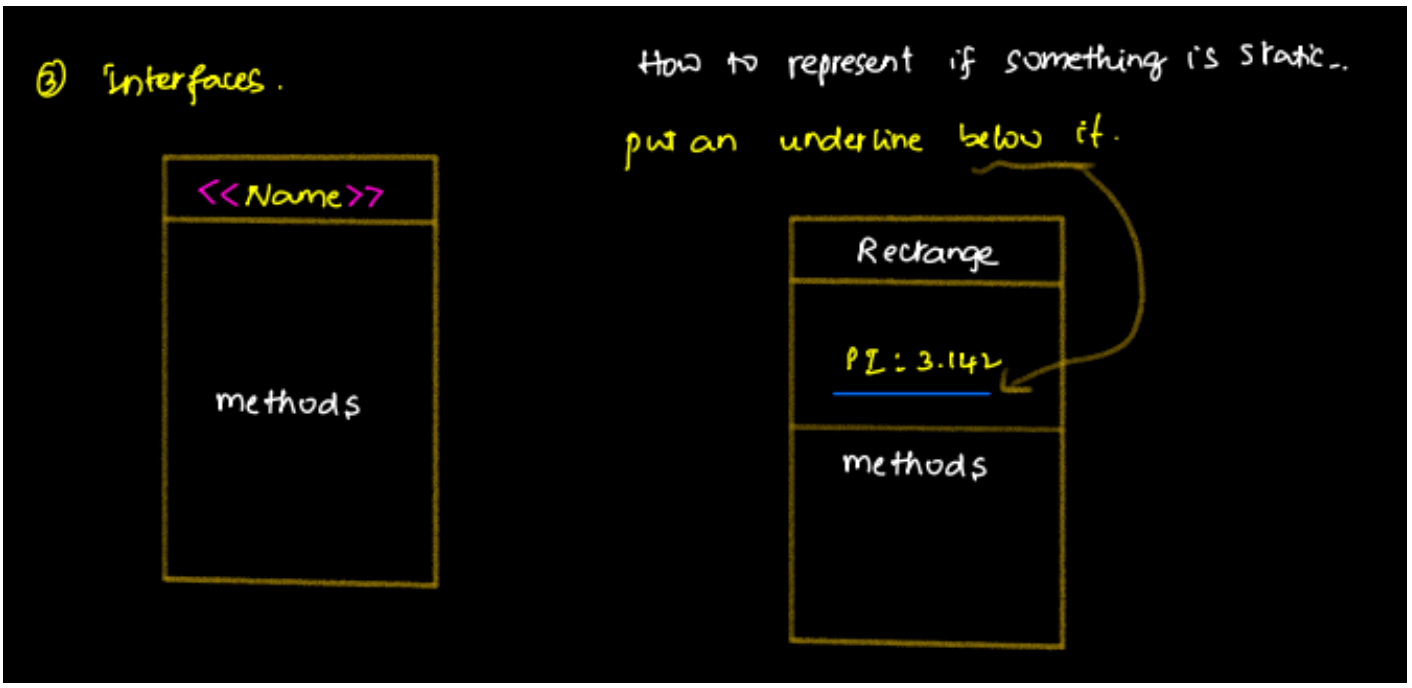
For Fields, the format is "access modifier" "name" : "datatype." We use "-" for private, "+" for public, and "#" for protected access.

For Methods, the format is "access modifier" "signature" : "return type," and the same access modifiers ("-," "+," "#") are used as suffixes.



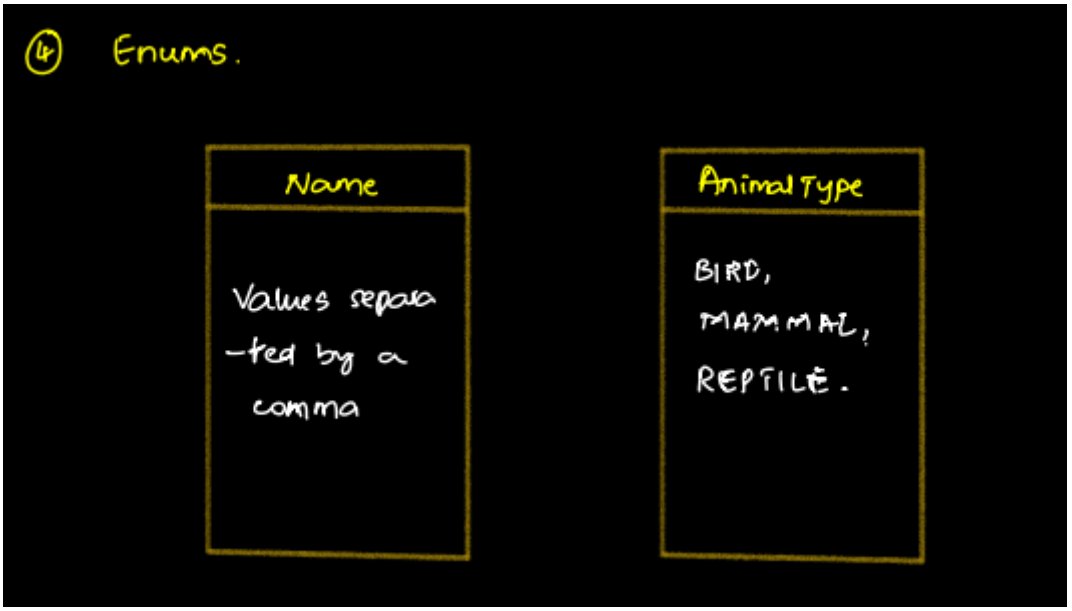
Abstract Class Diagram: Similar to a class diagram, an abstract class diagram specifically emphasizes abstract classes. Abstract classes provide a common template for other classes to inherit from, capturing shared characteristics.

Interfaces: It illustrates the relationships and interactions between different interfaces in a system. It provides a visual representation of how interfaces connect and relate to classes or other interfaces within the system. It consists of a two-row box. The first box displays the name enclosed in double chevrons **(<< name >>)**, and the second box is dedicated to listing methods. Interface methods do not require a suffix since they are static and final.



Enums: Enums (enumerations) are typically represented by a separate rectangle with the keyword "enum" at the top, followed by the name of the enumeration. Inside this rectangle, you list the possible values or constants that the enumeration can take

It comprises a two-row box, with the first row for individual values and the second row for values separated by commas.



In essence, a structural diagram helps developers and stakeholders visualize and understand the components, relationships, and organization of a system, aiding in effective system design and communication.

—----------Summary of the Module—----------

Creational Patterns:----------
Prototype Pattern: Creating new objects by copying an existing object.
Singleton Pattern: Ensuring only one instance of a class exists.
Factory Method: Centralizing object creation and business logic.
Abstract Factory Pattern: A factory of factory methods for creating related objects.
Builder Pattern: Creating objects step by step.

Structural Patterns:----------
Decorator Pattern: Adding behavior to objects dynamically.
Proxy Pattern: Creating a surrogate to control access to an object.
Composite Pattern: Treating individual and composite objects uniformly.
Adapter Pattern: Making existing classes work with others without modifying their code.
Bridge Pattern: Separating abstraction from implementation.
Facade Pattern: Providing a simplified interface to a subsystem.
Flyweight Pattern: Reducing memory usage by sharing data among objects.

Behavioral Patterns:----------
Decorator Pattern: Adding behavior to objects dynamically.
Proxy Pattern: Creating a surrogate to control access to an object.
Composite Pattern: Treating individual and composite objects uniformly.
Adapter Pattern: Making existing classes work with others without modifying their code.
Bridge Pattern: Separating abstraction from implementation.
Facade Pattern: Providing a simplified interface to a subsystem.
Flyweight Pattern: Reducing memory usage by sharing data among objects.

SOLID Principles:----------

Single Responsibility Principle (SRP): A class should have only one reason to change.
Open/Close Principle (OCP): Software entities should be open for extension but closed for modification.
Liskov Substitution Principle (LSP): Objects of a superclass should be replaceable with objects of a subclass without altering the program's behavior.
Interface Segregation Principle: A class should not be forced to implement interfaces it does not use.
Dependency Inversion Principle (DIP): High-level modules should not depend on low-level modules; both should depend on abstractions.