DEEP DIVE INTO DESIGN PATTERNS

SYLLABUS

SOLID PRINCIPLES

12 FOUNDATIONAL PATTERNS

STRATEGY

OBSERVER

DECORATOR

FACTORY

SINGLETON

FACADE

ADAPTER

COMMAND

PROXY

ABSTRACTFACTORY

NULLOBJECT

ITERATOR

This page have been intentionally left blank

SOLID DESIGN PRINCIPLES

Whenever we use object oriented programming, we have design principles that makes our code maintainable, readable, decoupled, testable and many other benefits.

We have a principle defined for each word, and these principles are invented by Robert Martin.

Single Responsibility Principle

" A class should have only one reason to change. "

Let's understand it more clearly, it can be confusing to understand it as one class should be able to do just one work. But this is not what it means.

By single reason, let's understand it with an example.

Let us take the example of an Order class of an Ecommerce app, and it has functions like addProduct() and getProduct(). Now, what do you think whose department this responsibility lies to? Maybe InventoryDepartment.

Now, this Order class also has more functions like calculatePricing() and generateInvoice(). Now, what do you think whose department this responsibility lies to? Maybe SalesDepartment.

So, there are multiple departments handling a single Order class.

Now, the reason word meant that, if in future we want to change this class, then there must be a single reason to change it, which means that there must be a single source to change it.

Here, in Order class, two departments can have their own demands of how they want to change something in this Order class, leading to multiple reasons of change, making our code complex and unmanageable, and may lead to conflict.

So, there can be different methods doing different things in a class but they must have a single source of change, a single reason of change.

And if we somehow found out that there are multiple sources of change, as in Order class, then we have to segregate the responsibility in multiple classes.

Open Closed Principle

" Software entities (modules, functions, etc) should be open for extension, but closed for modification. "

Whatever the code we write, it's not necessary to keep modifying it, instead just extend it.

Suppose we have written a function, and in future, we want to change it's behaviour, without actually modifying that function. Now, how can that be changed, if we do not touch that function? That's where this principle comes into place.

For example, suppose we have a PaymentProcessor class having process() function. Now, it can take two arguments, one for type of Payment as string, and other for amount as int. Now, suppose we want to add a new payment setup, and we pass that onto process() function as string. But here we have to modify the process() function, and we'll keep modifying it for all new payment setups, which keeps it open for modification in future, and violates our principle.

On the other hand, suppose we have created an interface IPaymentProcessor with process() function, and every payment

setup implements IPaymentProcessor and defines its own process() function. Then our code is not tightly coupled. And there is no need to modify the existing classes or the IPaymentProcessor to implement a new payment setup. So, we can add a new behaviour by extending it, rather than modifying it.

Liskov Substitution Principle

" An object (such as a class) may be replaced by a sub-object (such as a class that extends the first class) without breaking the program. "

If we are using a parent class in inheritance, and if switch that parent class with the child class, then your program should correctly work without any unusual behaviour.

How can we make that our code behaves the same way with child class as it did with our parent class?

Whenever a child class extends from the parent class, then all the methods of the parent class must be aligned within the child class as well. Child must have those behaviours or abilities that parent has.

If suppose we encounter a scenario when the child class does not have all the abilities of the parent class, then the child does not respresents the parent in a more specific manner, and one

way of dealing this can be to have a new parent class that aligns with the behaviour of that class, and using that separate parent class as per the behaviour of child class.

So, the parent class should only have those methods, that alings with the child classes.

Interface Segregation Principle

" No code should be forced to depend on methods it does not use. "

We should not have those methods in our code, that we do not use and still it has an implementation.

Whenever we have an interface with some methods, then we have to implement all of the methods of that interface in the class that implements it even if we do not use of all the methods defined in that interface. So, our code somehow has became dependent on those interface methods that it does not even use.

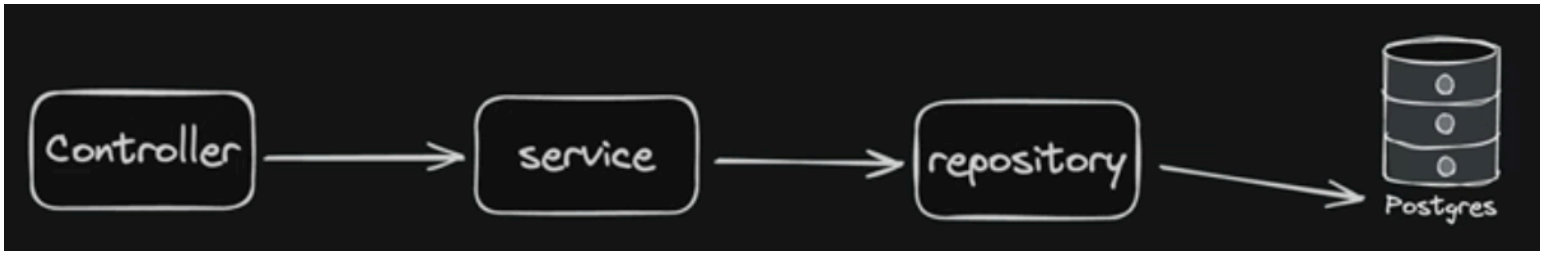
One way to do this is segregation of that interface that has more than one methods into separate interfaces each declaring its particular method.

Dependency Inversion Principle

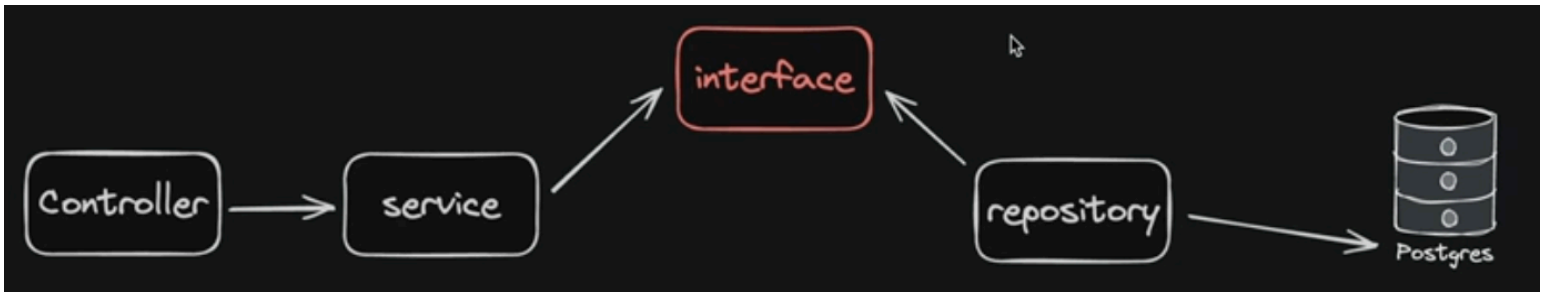
Through this, we can decouple our code, to make it easier for us to make changes to our code.

The principle states, " Any high-level module should not import anything from low-level module. Both should depend on abstractions (interfaces). "

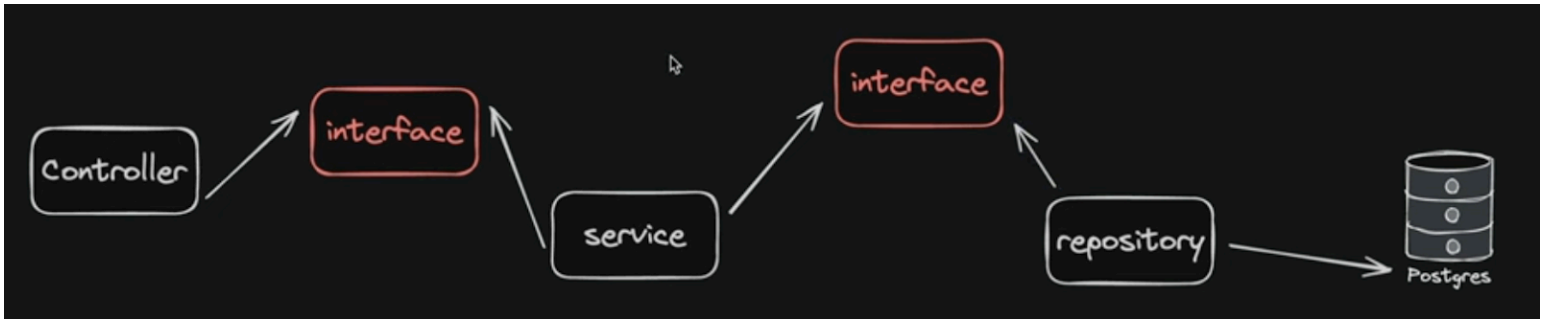
In our application, there are some high level modules such as bussiness logic, and there are some low level modules such as data layer, and they should not be depend such that our bussiness logic service depends on data layer, rather they should depend on abstractions.



The principle also states this, " Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions."

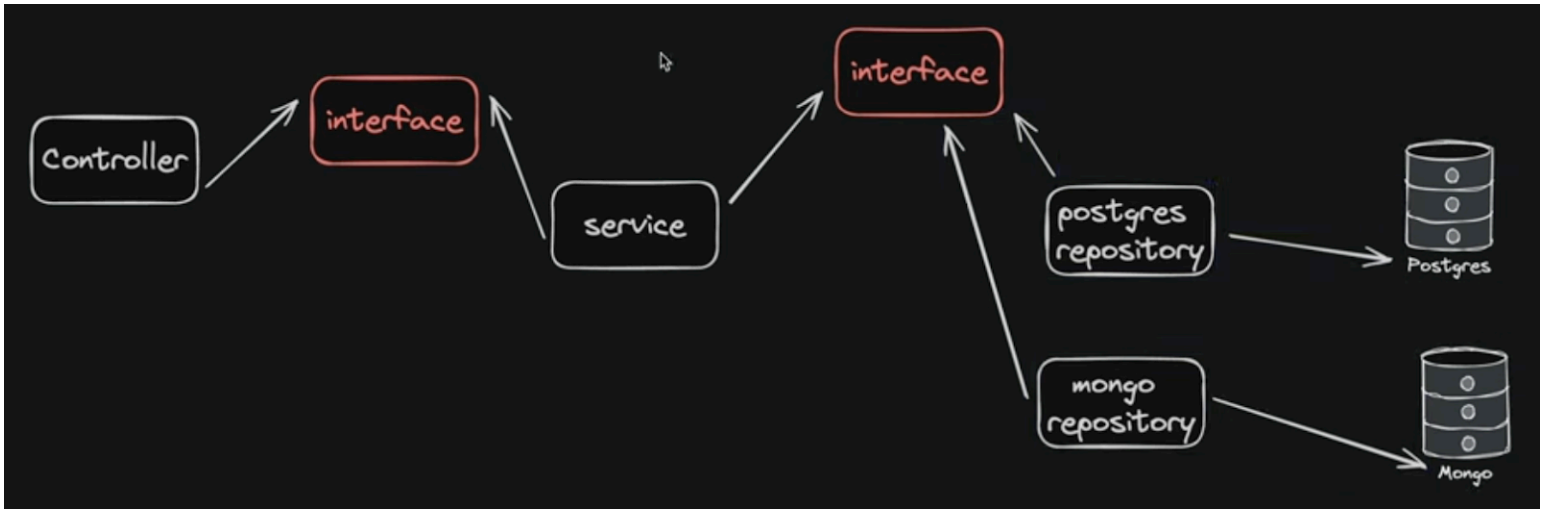


Our interface abstraction is not depend on either bussiness logic or data layer, rather they depend on the abstraction.



First, the dependency was from left to right, but now, we have inverted the flow of dependency, making our code loosely coupled.

Now, suppose we want do scaling, and add another database, because our code has been decoupled, we can easily add one using the common interface, without changing and modifying the existing code.



This page have been intentionally left blank

STRATEGY PATTERN

It's about using composition rather than inheritance.

" The Strategy Pattern defines family of algorithms, encapsulates each model, makes them interchangable. Strategy lets the algorithm vary independently from clients that use it. "

Strategy pattern says that you may be having a set of algorithms that you want to use and what I would do is encapsulate each one of those algorithms and make each one of them interchangable. You may have algorithm A, B And C. What Strategy pattern is saying that you can just plug and play, and use any of it howsoever you want.

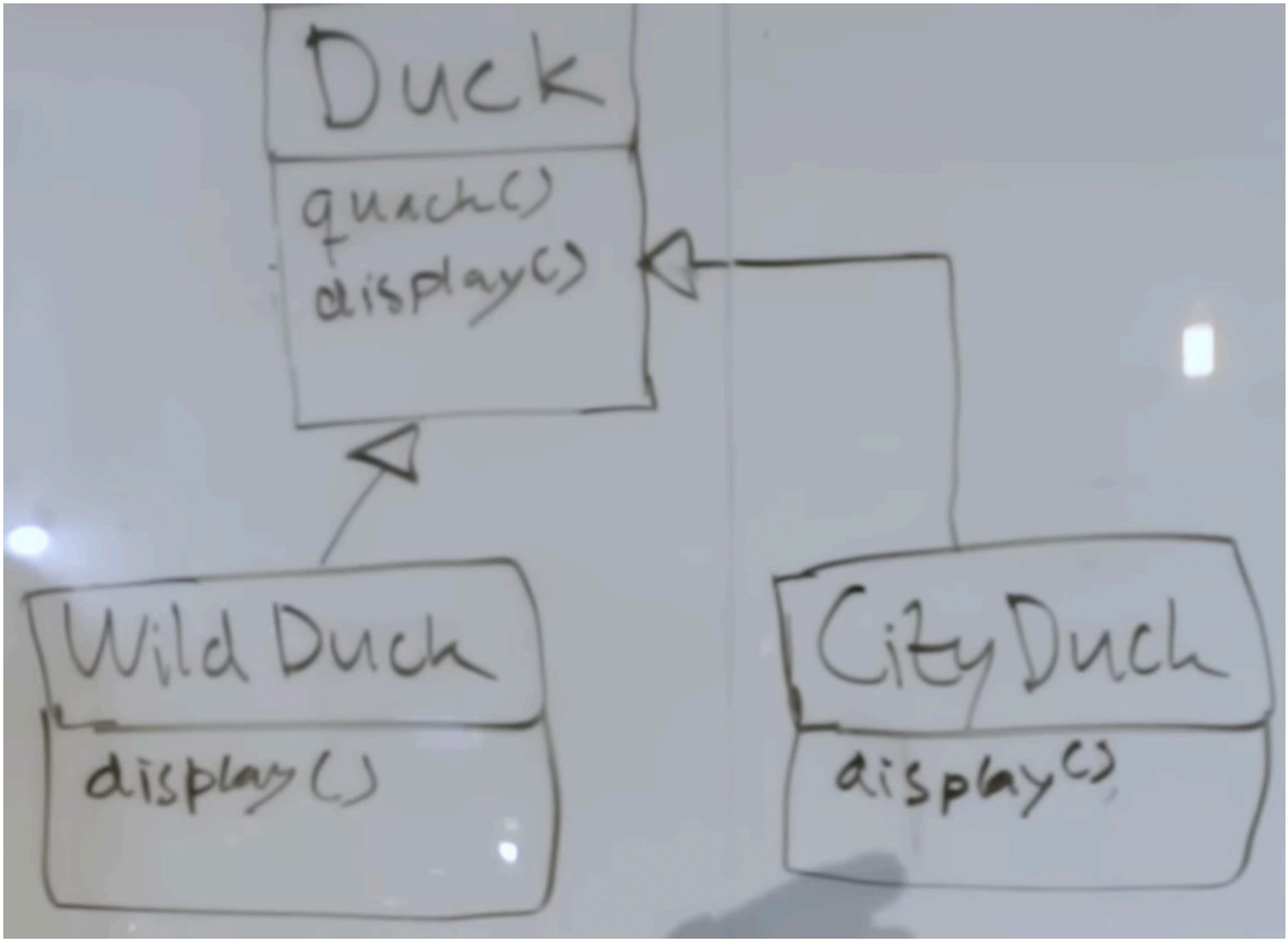
Then it says, strategy lets the algorithm vary independently from the clients that use it. So what we are doing here is decouple the algorithm from the one using it. So it can vary between from algorithm A, B and C. So whosoever is using algorithm A, B and C, that client does not have to vary if one of the algorithm varies.

In simple words, by vary, if we want to change one of the algorithms, the context of the algorithm, what the algorithm actually does, then we don't necessarily have to change the client at the same time. So, whosoever is using that algorithm is not forced to change as we change the algorithms.

For example, we can think of a list, if the list has a sorting algorithm, built into it, we can change the sorting algorithm. But if we use the strategy pattern, then the sorting strategy can vary independently from the list. The implementation of list is independent, but then we inject the sorting strategy into it.

Here's a example, there's a duck, this duck class is the super class, and the intention is that other classes should inherit from this duck class. Now, we have 2 types of ducks, a wild duck and a city duck, probably because a wild duck lives in the forest, and a city duck lives in the city :)

So, a wild duck is a duck, which means, a subclass duck inherits from superclass duck for implementing their own version of the methods, which means we are overriding the methods in inheritance such as display() method. However, there are methods which have the same behaviour for all ducks, that is their quack() behaviour.



So, it seems perfect, why do we need a strategy pattern then?

It's like when requirements change, our system will have to change overtime. And the current design may not be appropriate for our requirements.

So, suppose we have another method fly() by assuming that ducks do fly, in the superclass.

And suppose, that another subclass Rubber duck inherits from Duck. Now, rubber duck has it's own display method.

But here, the argument here is that, rubber ducks should not fly, so we have to override fly() method in rubber duck class.

Now, let's say we have Mountain Duck as a subclass inheriting from Duck.

And here, the argument is that, Mountain ducks have a different fly behaviour than Duck fly behaviour.

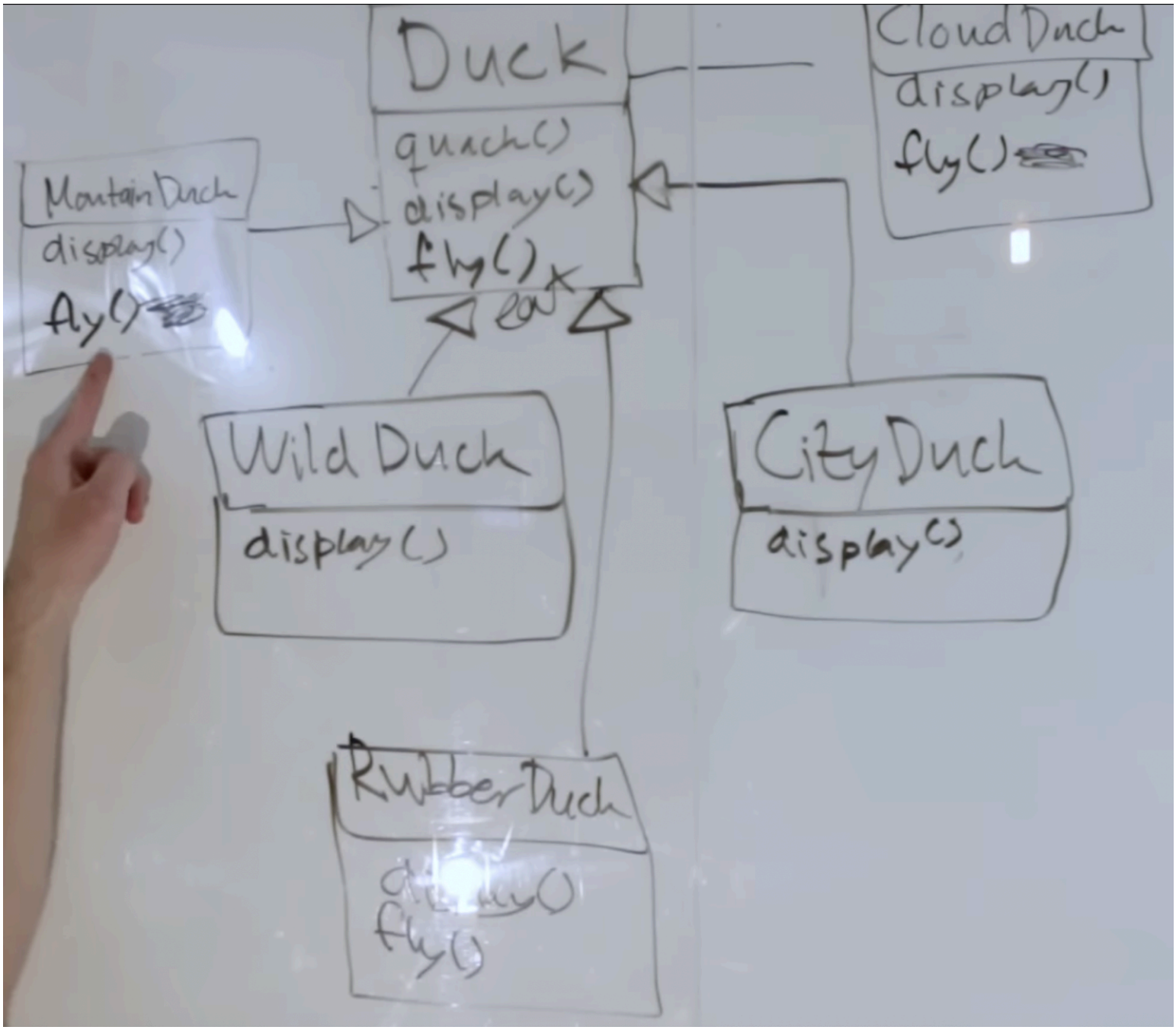
Now, we have another duck named Cloud Duck as a subclass from Duck also having it's own fly behaviour.

But, the Mountain Duck and the Cloud Duck has the same kind of fly() behaviour.

And we can see that the fly() method code has been duplicated in both Mountain and Cloud Duck subclasses.

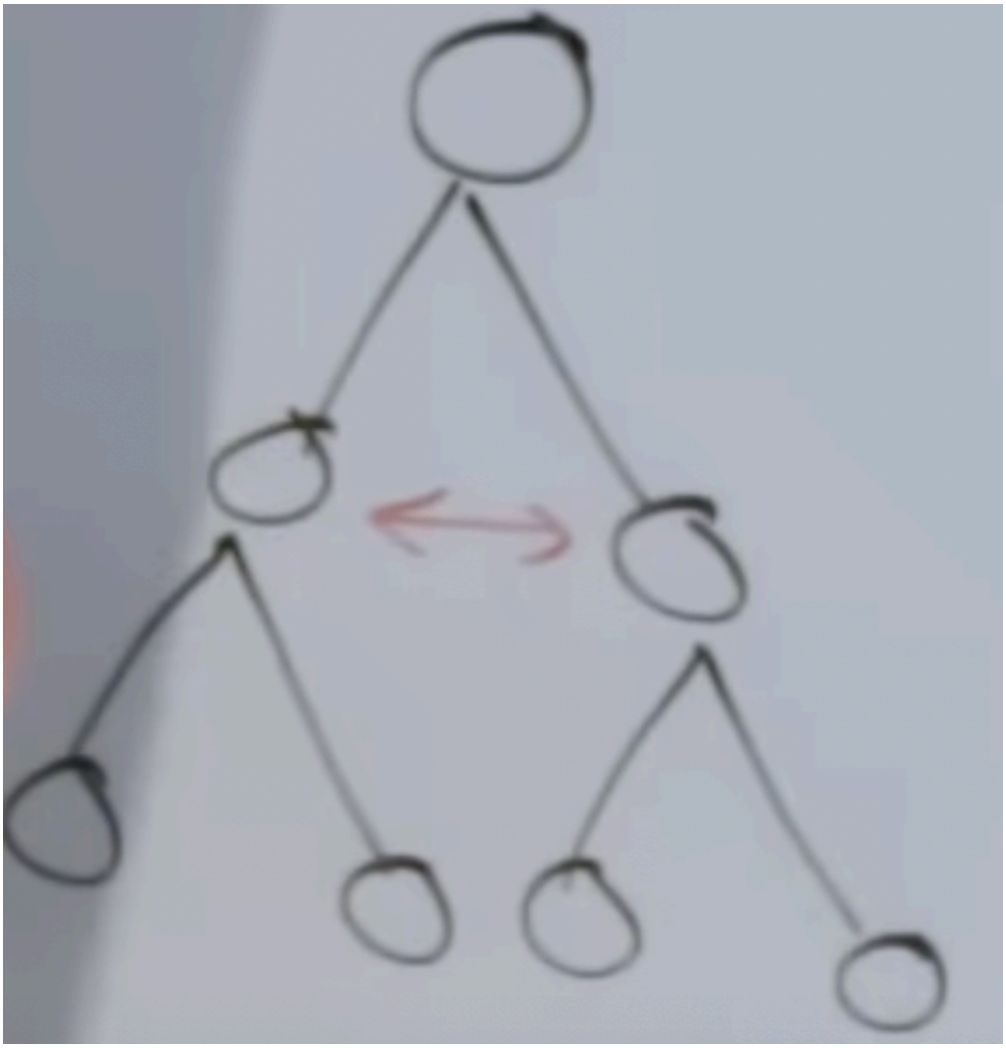
Maybe to fix duplicate behaviour, we can have another superclass with fly() method that Mountain and Cloud Duck has.

And what if we also have eat() method for example, and it is different in these duck sub classes, but duplicate in some other sub classes.



So, the idea behind inheritance is that as long as the behaviour is shared downwards, it's good.

But if we want to share the behaviour horzontially among classes, there's no way.



" The solution to problems with inheritance is not more inheritance "

So, no matter how we build it, there are indeed some scenarios where we can't build it heriarchially.

And we'll end up with duplicate behaviour and methods at a same level of inheritance.

The solution to these are composition.

So, what strategy pattern says that we have algorithm for fly(), and what we are realizing is that we can't build a hierarchical solution to share code between different uses of the different algorithms, so we have to extract the algorithms and say that these are clients. So, we have all subclass ducks as clients, and they make use of these algorithms for the fly() method, and they these algorithms must be able to vary independently from other aspects of the client.

So, there are essentially multiple ways of approaching strategy pattern but we'll go through one way.

So, what we are doing is creating separate interfaces for each behaviour, one for fly(), one for quack()

We can also do something like have one class which defines a particular behaviour using certain interfaces, like using generics.

So, when a Duck instantiates a class that defines the behaviour using decopled interfaces, then composition is achieved.

These behaviours are nothing but algorithms, which we have composed and combined them in separate decoupled classes using interfaces, which every ducl class can use.

So, we can take these two different algorithms, meaning these two different strategies, meaning these two different behaviours,

and use them to compose new things with different independent clients.

To put simply, we created an interface for each behaviour, suppose fly() interface, and we have multiple classes that molds and defines the fly() behaviour by implementing that fly() interface. For example,

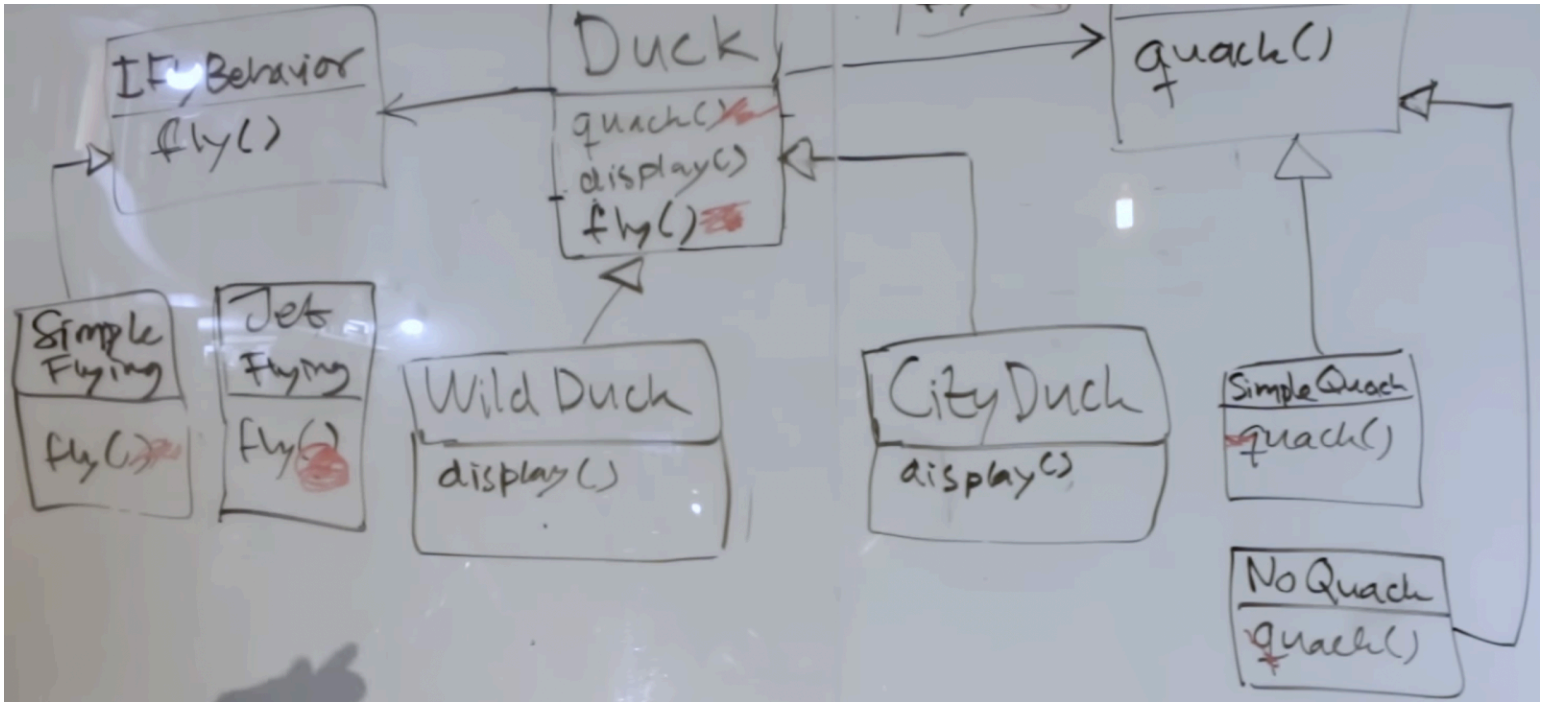
```
interface IFlyBehaviour{
void fly()
}

class SimpleFlying implements IFlyBehaviour{
 public void fly(){}
}

class JetFlying implements IFlyBehaviour{
 public void fly(){}
}
```

Now, these two flying behaviours can be used in various Duck classes, irrespective of how the client is.

Hence, duplicacy does not occur here, making the code simple and clean.

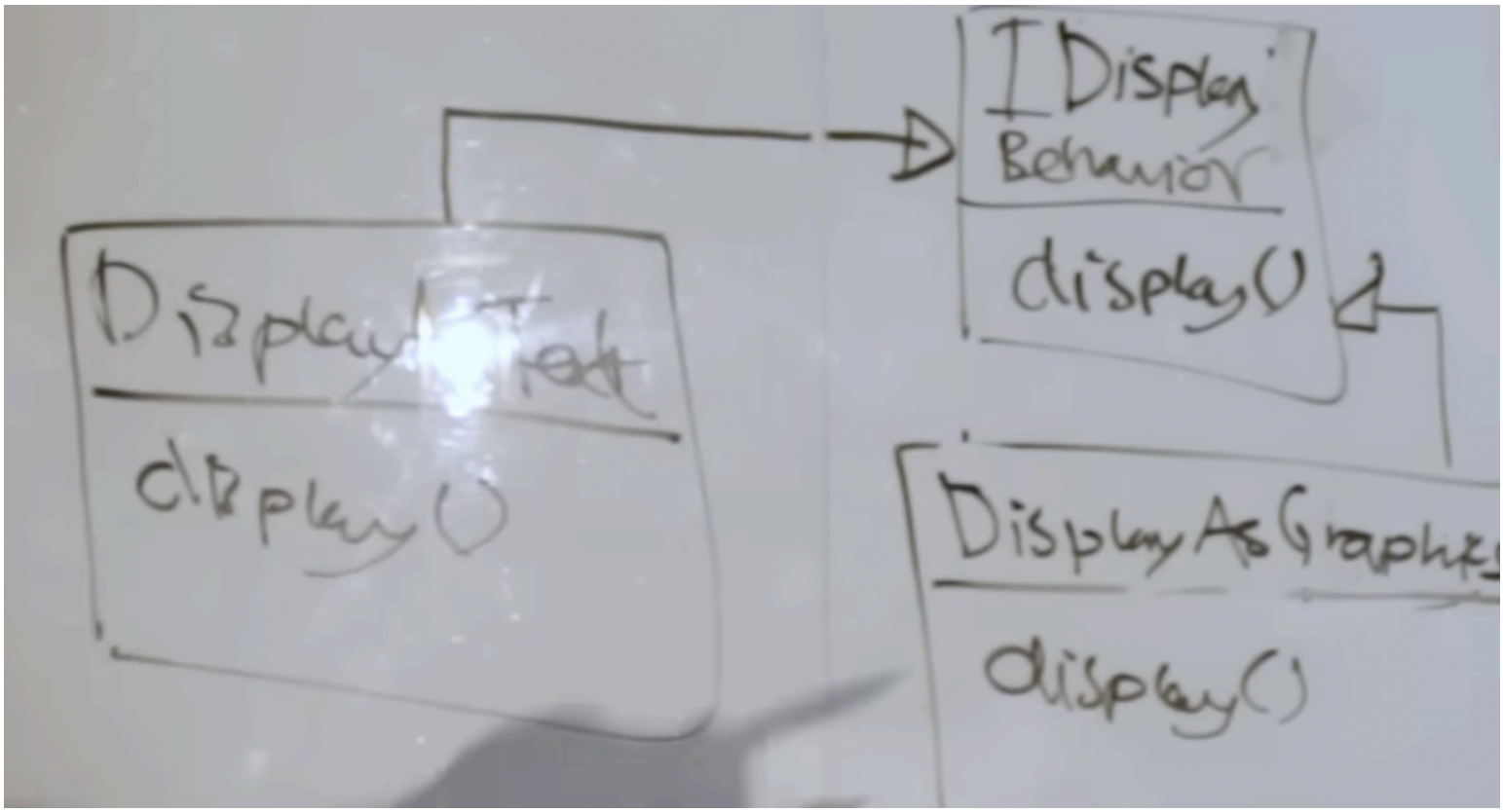


So, we can do this for every behaviour, extract that behaviour as an separate interface, and let multiple classes define their own ways, making them reusable and flexible.

The main intention of using Strategy pattern is reusability of independent components, or behaviours, or algorithms. If these components are not being reused, it makes no sense to actually separate the behaviour in a class implementing its interface, it'd only increase complexity that way.

For example, City Duck And Wild Duck both has their own display methods. And if we are creating something like CityDuckDisplay or WildDuckDisplay class that implements Display interface, just only because this behaviour is being applied to one and only one duck, then strategy pattern should not be used this way.

But if we have behaviours that are reusable, and actually defines a certain difference in thier behaviour, such as DisplayGraphically or DisplayText, then strategy pattern can be used.



" So, it seems like, with this example, we are somehow arguing that, interfaces would be more appropriate than subclasses. "

Now that we have these concrete strategies, there is no need to override any of the behaviour, because we can simply use our composed independent behaviours.

So, we had so many types of ducks, from city duck, wild duck, cloud duck, to rubber duck kind of things. But these are no longer named classes of a system. They are a particular

configuration of instances of behaviour. So, given a combination of different strategies, and combination of different algorithms. We happen to call some combinations different things.

But what we haven't talked about here is Dependency Injection. That is only possible if these behaviours are injected into instance of a Duck, and not hardcoded in the class, and we can't do that anymore.

Now, let's look at some pseudo code very very quickly,

```
class SimpleDuck{
IFlyBehaviour fly = new NoFlyBehaviour();
IQuackBehaviour quack = new QuackBehaviour();
IDisplayBehaviour display = new DisplayGraphically();
}

class RubberDuck{
IFlyBehaviour fly = new SimpleFlyBehaviour();
IQuackBehaviour quack = new NoQuackBehaviour();
IDisplayBehaviour display = new DisplayText();
}

SimpleDuck duck = new SimpleDuck();
RubbberDuck rubber = new RubberDuck();
```

So, basically here, we've decoupled and tried to reuse the behaviour by creating different classes without inheritance.

But this is not the proper way of strategy classes, it's not flexible enough. How many classes would you create? It's repeatable.

Better way is letting constructor build different behaviours from the same class Duck.

```
class Duck{
 IFlyBehaviour fly;
 IQuackBehaviour quack;
 IDisplayBehaviour display;
 public Duck(
                IFlyBehaviour fly,
                IQuackBehaviour quack,
                IDisplayBehaviour display
            ){
  this.fly = fly;
  this.quack = quack;
  this.display = display;
  }
}

Duck duck;
duck = new Duck(
                new NoFlyBehaviour(),
                new QuackBehaviour(),
                new DisplayGraphically()
                ); // simpleDuck
```

```
duck = new Duck(
                new SimpleFlyBehaviour(),
                new NoQuackBehaviour(),
                new DisplayText()
                ); // rubberDuck
```

This is the perfect, reusable, composable and clean code using Strategy pattern.

Now, you can define a fly(), quack() and display() method for the Duck class, which will call the method from it's behaviour classes.

```
public void Fly(){
 this.fly.Fly();
}
```

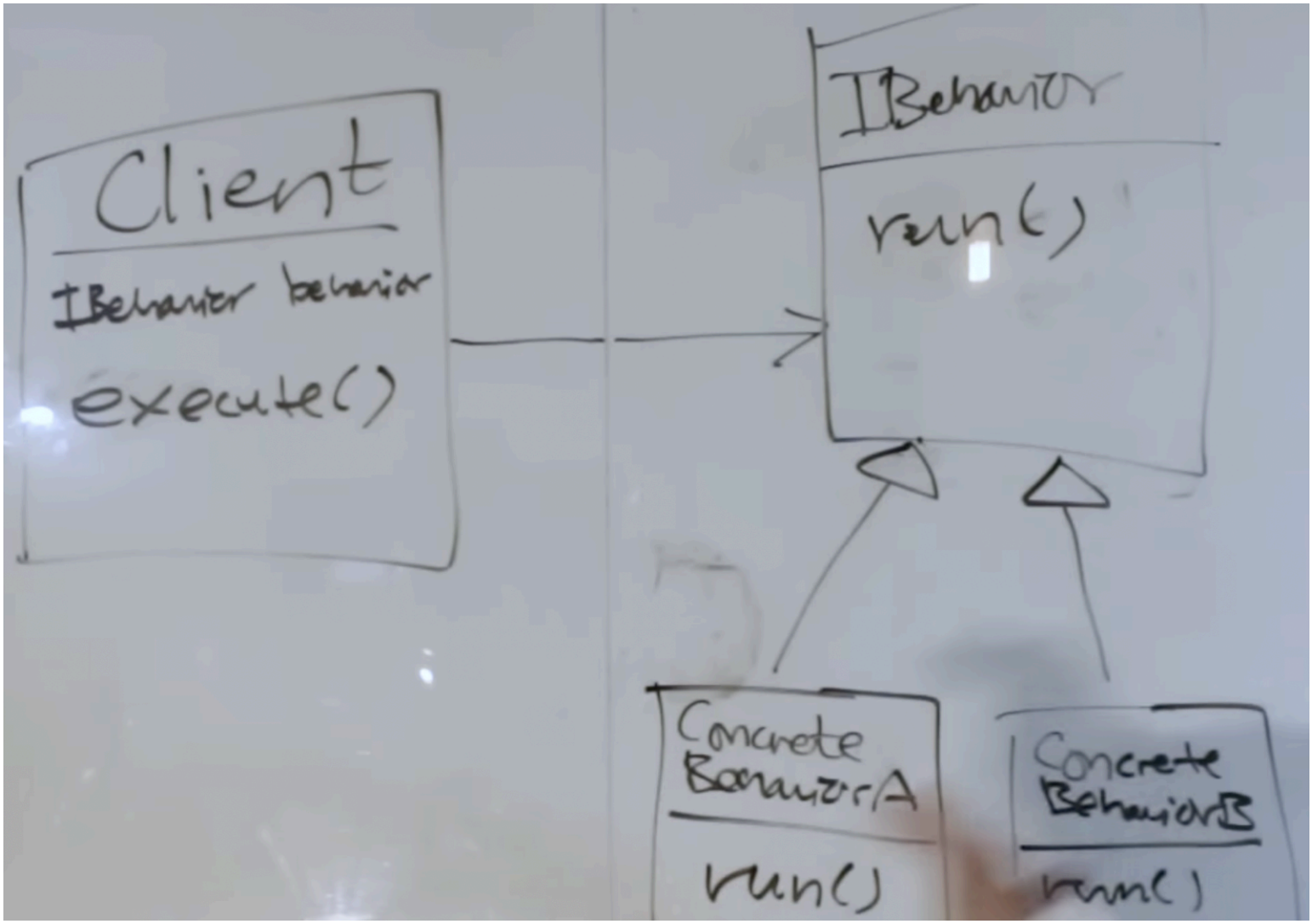
So, you pass in a behaviour, and then you execute that behaviour within the class.

Now, ducks vary independently of it's behaviours.

Let's look at the generalized UML Diagram of Strategy Pattern,

We have a Client named Duck, and that client has IBehaviour, which can be implemented through its interface or abstract class, but if it's an interface, it can be implemented by concrete behaviour classes A B and so forth. And our Client has a

reference to IBehaviour instance. Now, a method from the Client can call the method of IBehaviour's concrete behaviour classes.



Now, the definition would make much more sense. Let's go through it again.

" So, the strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangable. And thus, strategy lets the algorithm vary independently from the clients that use it. "

This page have been intentionally left blank

OBSERVER PATTERN

Imagine that you have two things, and one of these things, changes state, and overtime it's internal state changes.

Now, the other thing wants to know about the state that thing has.

Look at the example of a Weather station, that measures weather data, and does some measurements on data which may or may not have changed.

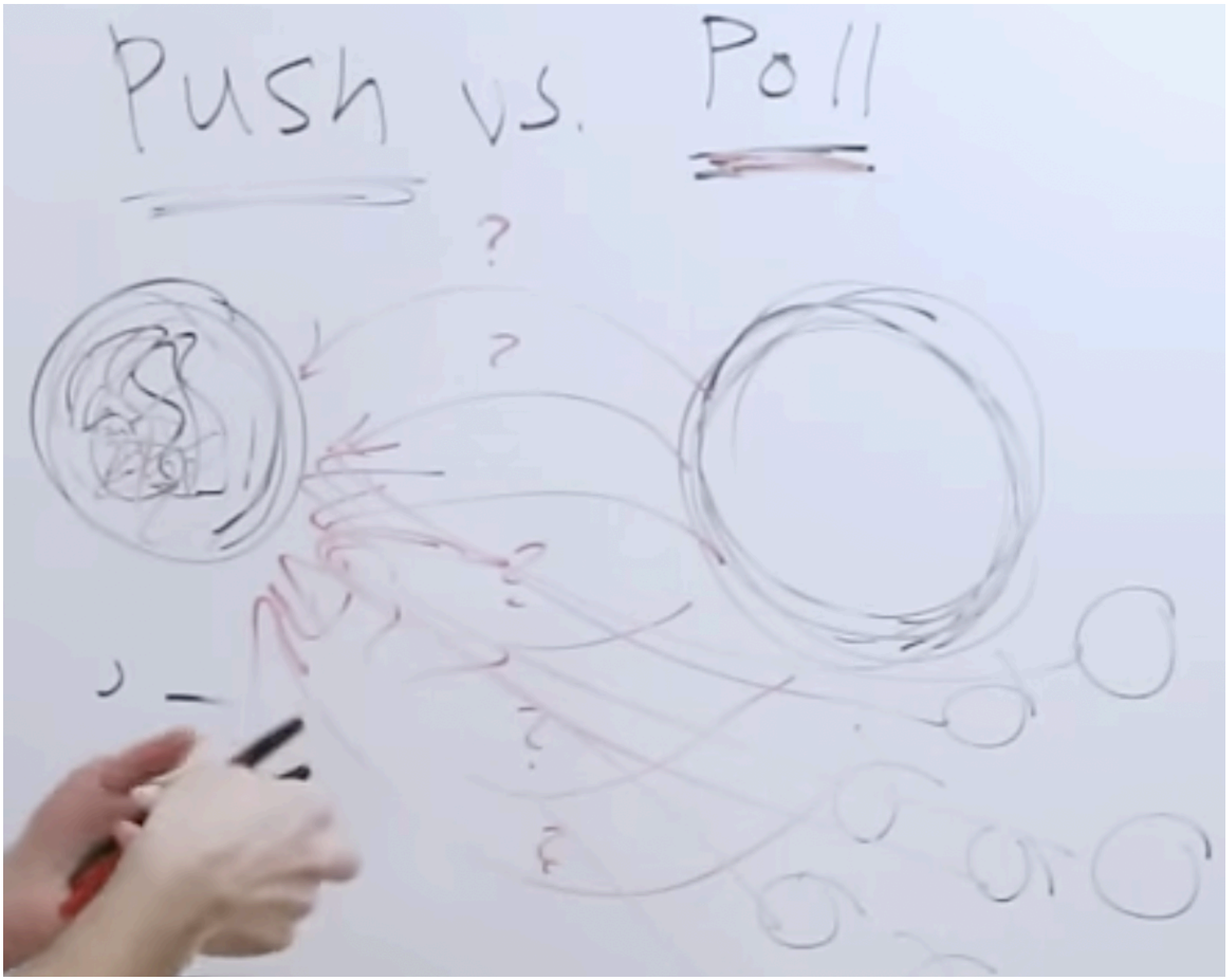
Suppose we have an RSS client, a feed, and it wants to know when an article has been published.

So, observer pattern is essentially about Push vs Poll. It's about moving from Poll architecture to Push architecture.

It's used more in like networking architectures and web services and kind of things.

One of the obvious thing that comes to mind is that the observer can just ask the other object straightaway whether you have changed?

And it'll keep asking it again and again? And suppose we have tons of observer objects all wanting to keep track of the state of other object, that'd not be practical to have so many API calls just to ask whether the other object has been changed or not? And if we don't even know the frequency with which we have to ask for change?



This is called Poll. Have you changed your changed?

The point is, we're asking for the data, before we actually know that we have the new data. Which is not really good.

Instead of that, the observer pattern helps us with this. It focuses on that object which changes it's state to respond to the observers when change happens.

So, this guy which changes, should be responsible for telling all it's subscribers when it has changed.

Rather than asking over and over again, we just send the response to subscribers when the state has actually changed.

So, it pushes the notification, of how it has changed to multiple subscribers.

In essence, this is the observer pattern.

Wait, this changing object has some knowledge about who his subscribers are.

Somehow, before we can do any pushing, one of the subscribers needs to register to the subject as a subscriber.

So, the subscribers being registered are Observers, and to which it has been registered is Obseravable.

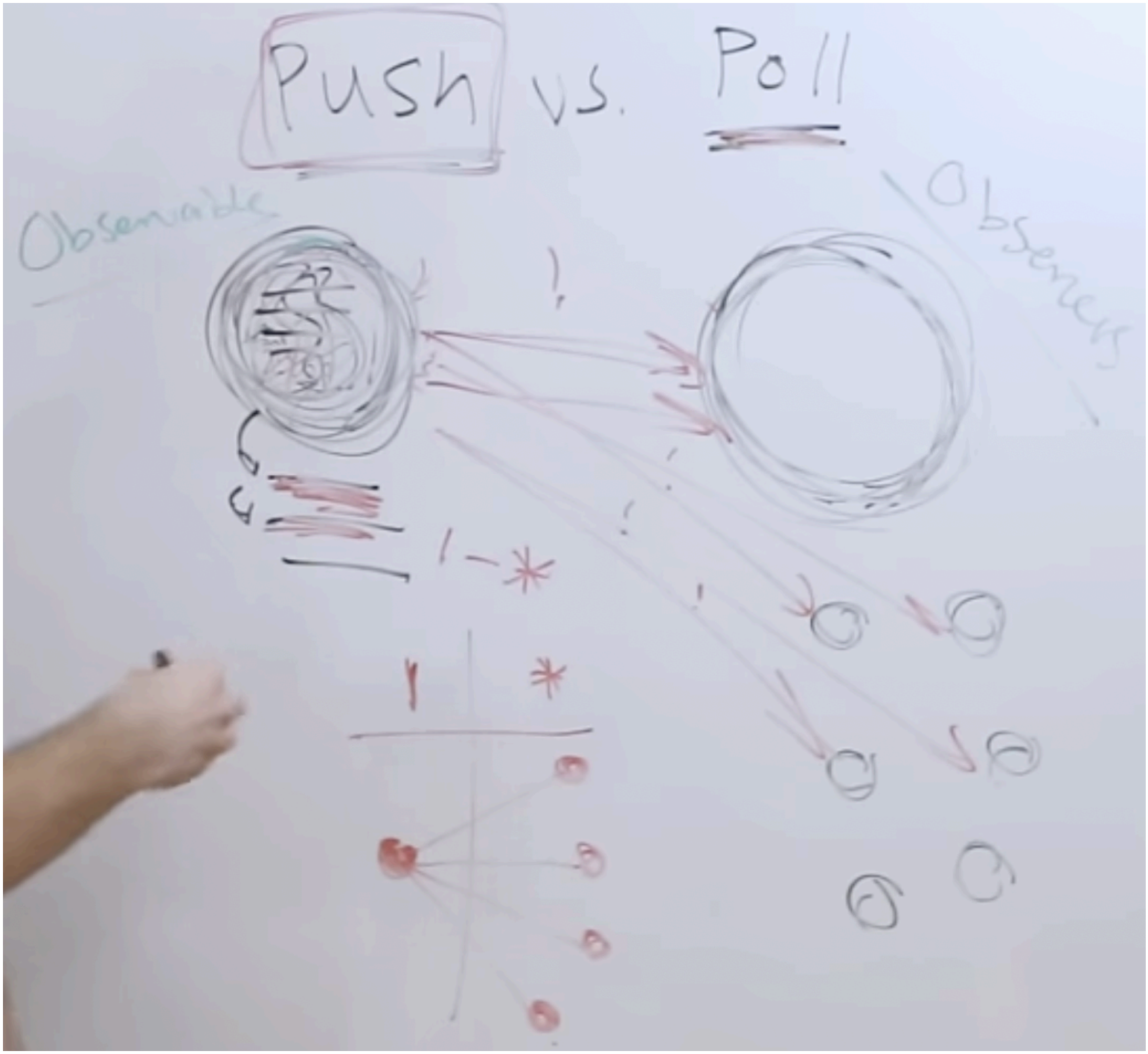
So, observable is being observed by observers. When the observable changes, the observers are notified.

Observerable is something that can be observed, can be seen, can be known, by some others, that are looking at, that thing, as observers.

Let's look into the actual definition,

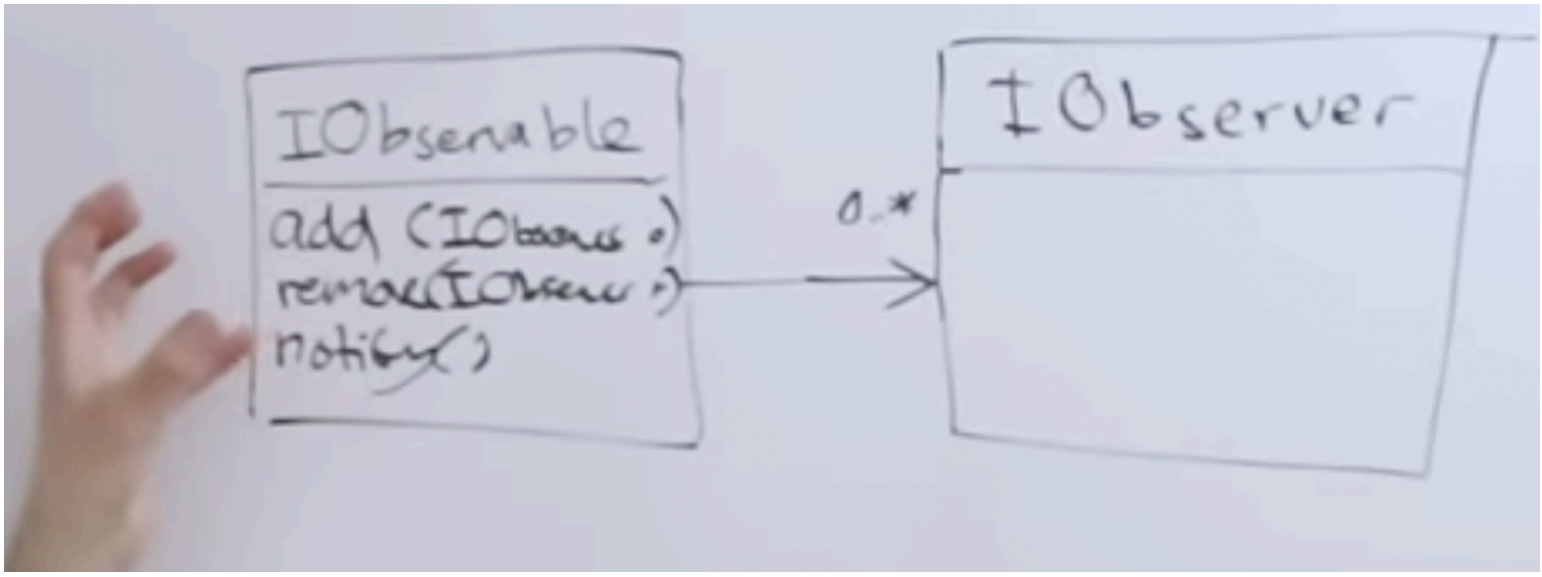
" The observer pattern defines one to many dependency between objects such that when one object changes state, all of it's dependencies are notified and updated automatically. "

The one to many may have relation such that one observable has many observers, and when the observable changes, all dependent observers are changed automatically.



Let's look into the UML diagram below,

In our diagram, interfaces are represented with an convention prefix " I ".



So, the thing is that the observable needs to be able to keep track of who wants to know when something changes, and the things that want to know are observers. So, somehow we need to be able to add instances of observers to an instance of an observable. So, this is why we have add and remove methods. We can also call these as register and unregister observer.

And the method notify is essentially the method we can call to broadcast the fact that htis has changed to all of the observers.

So, essentially observable has some kind of collection that contains a bunch of observers. It can be an array, or a hashmap.

Let us go into the actual implementations now, so we have concrete observable, and concrete observer.

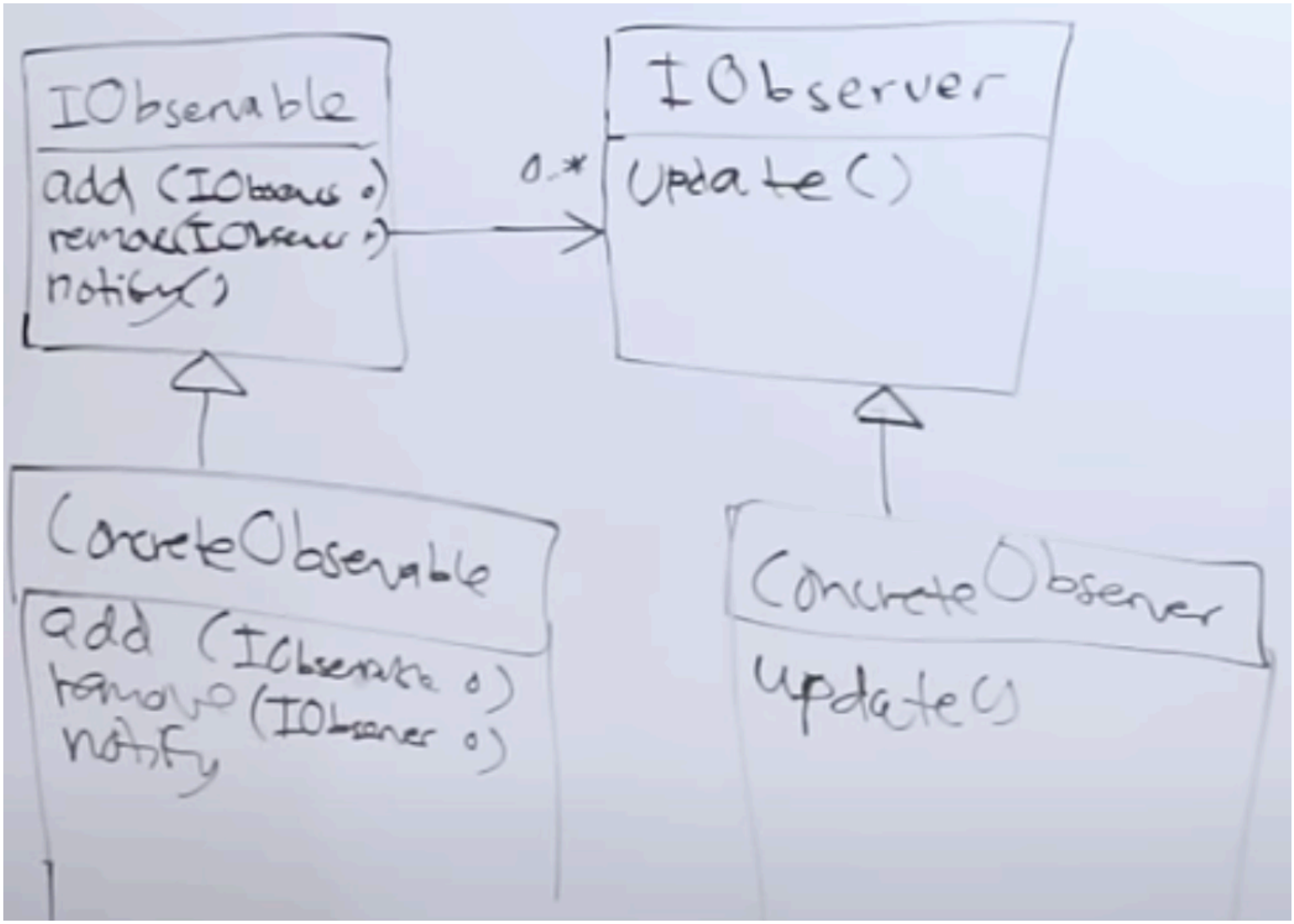
So, the ConreteObserver implements the IObserver interface, and the ConcreteObserverable implements the IObservable interface.

But hey, we also to define the contract of how IObserver has responded back to IObservable.

So, IObservable should be able to respond to a call to the method update() in IOberserver.

So, we instantiate observers, you add them using add() method, and then whenever you call notify, whoever calls notify, the observable instance will then call update() on all of the instances of the IObservers, and then ConcreteObservers that have defined those update() functions in their own way will work accordingly.

Because these are concretions, and they implement the interface, we can just copy the signatures of the interfaces because ofcourse they have to be implemented in the concrete classes.

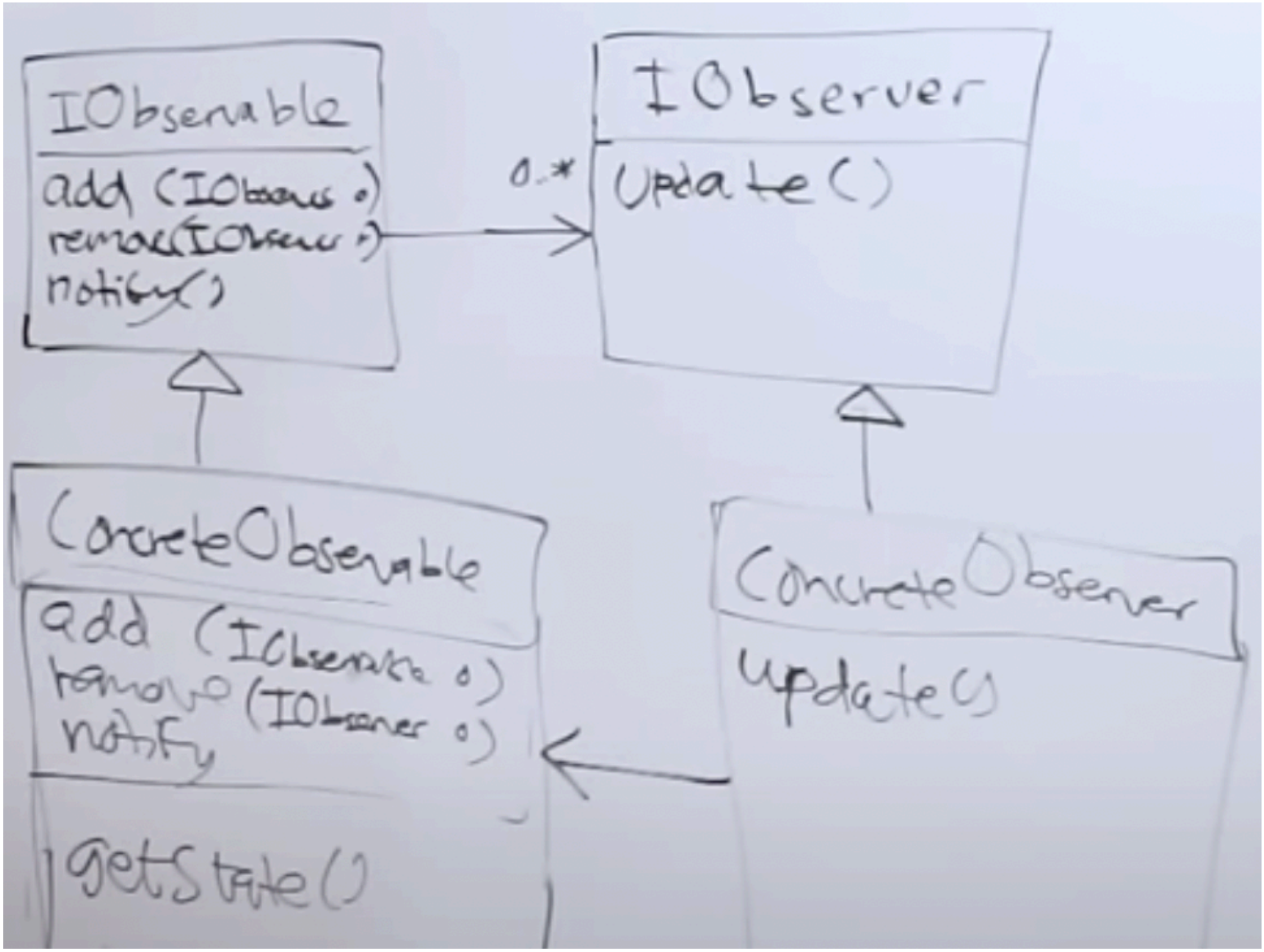


Observer pattern is a very suitable architecture for chat system, so if you think about it, you have a chat room, right, which is an observable, and then you add in observers to the chat room, and then whenever somebody broadcasts a message, right, we save that message into the observable, and then we call notify so that all of the observers that are connected to the chat system can get that latest message that has been broadcasted by one of the users.

Now, we're going to put an arrow from ConcreteObserver to ConcreteObservable, but it won't make sense, because when we

talk about design patterns, we want to increase abstraction and usually we want to have dependencies between abstract things and not between implementations. We want to have dependencies between interfaces and not between concretions.

But here we actually have a concretion that has a relationship with another concretion, so why? The idea is that when you instantiate a ConcreteObserver through the constructor, you pass the ConcreteObservable to it. So, you pass that ting that you want to observe, so it kind of goes two ways. So, the observer is passed into an observable, via the add() method but then we take the observable and pass it into the observer as well upon it's construction.



When we do this two ways, that means that the ConcreteObserver has a reference to the ConcreteObservable that's observing it, and if it has that, when we call update(), we don't have to pass anything here in the update() method anymore, because if you think about that, the upadte method if it has no arguments, then we're just saying that you were interested in knowing when this observable changed, and that's it. But if we're not passing anything here, then how would you know what has changed, or how would you know what to read, how would you get the information?

new ConcreteObserver(observable);
// passing the instance of observable that we want to observe

The code above is just creating the observer instance, it's not registering us as an observer right now, unless add() method is invoked on those observers by the observable.

Let's look at an example now,

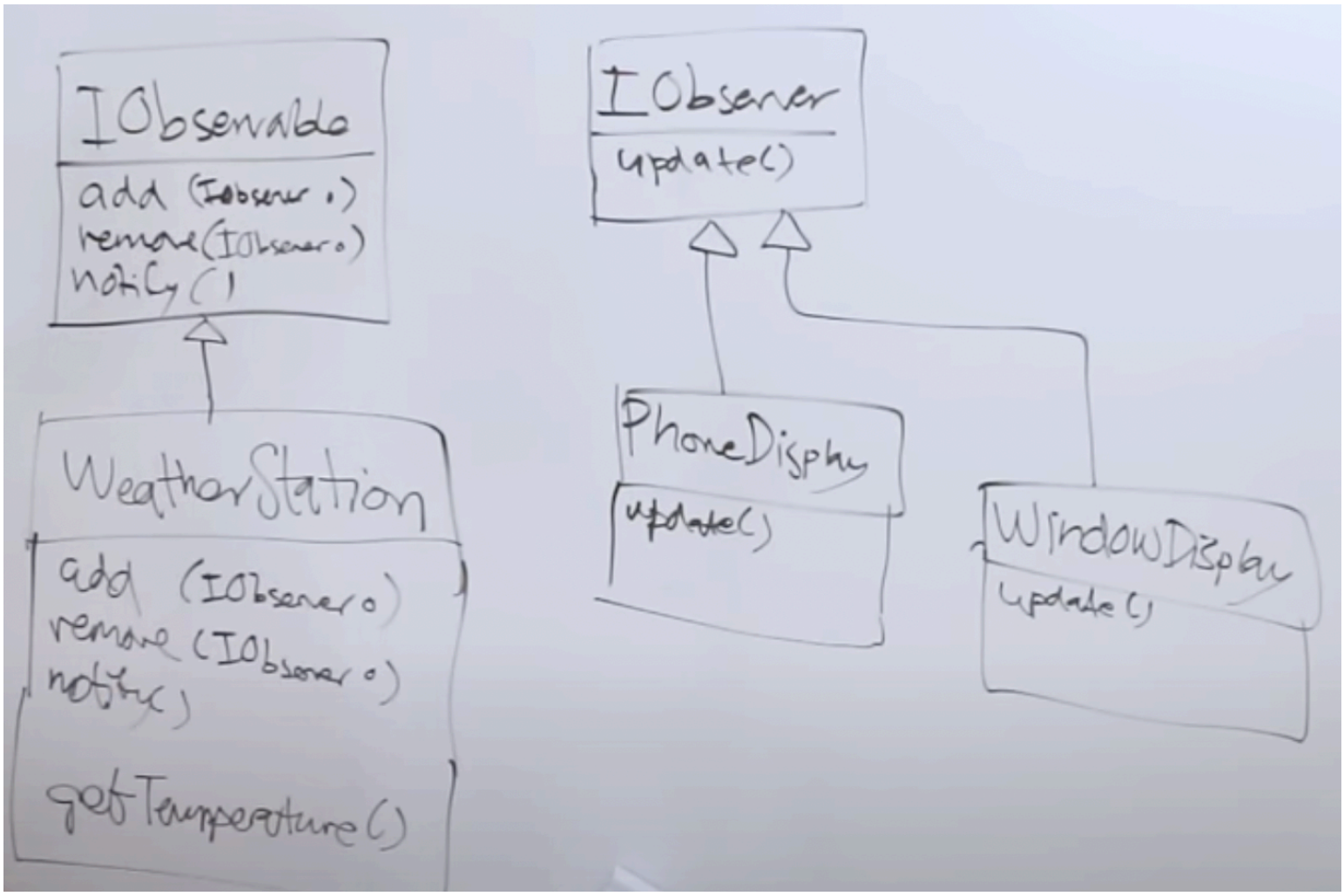
Let's say that you have some kind of weather station and this weather station receives updates by sensors about the change in the environment, and through observing the environment, it updates its own state of its measurement. So, it's temperature and its humidity and etc.

Let's say you have the measurement station that measures your temperature, just think about the room that you're currently in, just imagine that you have some kind of sensor somewhere and that's the weather station, you have some sensor but that sensor doesn't display anything, right, that just measures the temperature. So, whenever the temperature changes, it pings right, it sends a message to whoever you've registered as an observer that might be your cell phone for example, or a physical display next to your desk

Suppose we have two displays, that are the observers, and they want to observe the observable and the obserable is the weather station, so let's draw this,

So, WeatherStation class is going to implement the IObservable interface, and the PhoneDisplay and WindowDisplay classes are going to timplement the IObserver interface.

Now, somehow these observer guys need to be aboe to inspect the state of the weather station, that is observable. In other words, whoever has access to an instance of a weather station can ask the weather station, what it's current temperature is. Weather station has sensors so it's storing the temperature information.



WeatherStation station = new WeatherStation(); // observable
PhoneDisplay display = new PhoneDisplay(station); // we passed an instance observable to the observer
station.add(display); // we passed the observer to the observable

The display must be able to look at the weather station and figure out what the current satate of the weather station is, because it's pulling, and when it exacly knows that the state has changed. So, it needs to somehow access the new data, and for we then do what we need to to make use of observer pattern. And then we are adding the observers to the observable as per our observer pattern that as soon as observable's state is changed, it calls the notify() function, which calls the update() method in all observers.

Let's look into the observer pattern code then,

Let's first define the general interfaces,

```
interface IObservable{
add(IObserver);
remove(IObserver);
notify();
}

interface IObservable{
update();
}
```

Let's look into the concrete classes with the example of a weather station.

```
public class WeatherStation implements IObservable{
 ArrayList<IObserver> observers;
 int temperature;
 WeatherStation(){
 observers = new ArrayList<>;
 }
 public void add(IObserver observer){
  observers.add(observer);
 }
 public void remove(IObserver observer){
  observers.remove(observer);
 }
 public void notify(){
   for(int i = 0; i < observers.size(); i++){
    observers.get(i).update();
   }
 }
 public int getTemperature(){
  return temperature;
 }
 public void setTemperature(int val){
  // some logic that changes the temperature
  // basically it is changing state here
  temperature = val;
```

```
  notify();
 }
}
```

Essentially, the weather station has a collection of observers you iterate over, and then you call update on each of those observers, and this is the heart of the observer pattern.

And we're using the temperature which is acting as the change of state here.

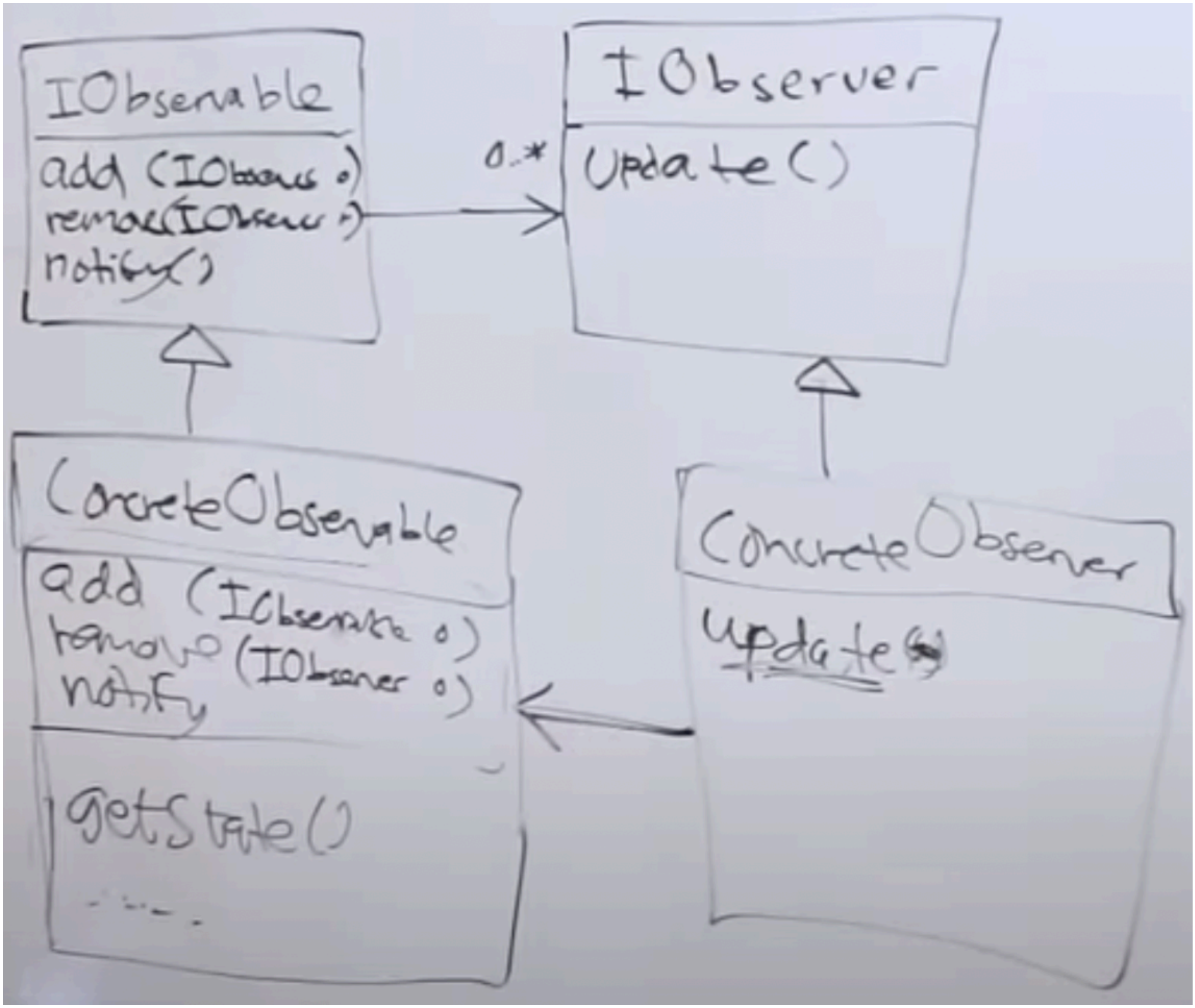
Let's look into the observer now,

```
public class PhoneDisplay implements IObserver{
 WeatherStation station;
 PhoneDisplay(WeatherStation station){
  this.station = station;
 }
 public void update(){
  int newTemperature = station.getTemperature();
  // some logic that updates the display as per new value
 }
}
```

We're passing in an depedency injection through the constructor and then setting that onto the instance level, that is, the weather station in the phone display. We're coupling to the concretion weather station class in phone display, which makes sense

because phone display here is only for weather station. Here, a phone display isn't a phone display that displays anything, but we're modeling a display that displays only the weather station data. Thus, it can couple to the concretion weather station if we want to model a display that can display anything.

For example, if phone display is a kind of subscription, like an article reader, and we know when we call update, there might have been a new article in the database of article rader, so I need to update my list of articles that have been here for the display. So, basically, when update() method is called from the observable to each of observers through observable's notify() method, the observers know that they also need to react to this, because that's the reason they have subscribed after all. They want to be able to react to this and how to get that information from the weather station, which is through the reference we have already passed to the phone display.



If we look back into the UML diagram, it makes perfect sense now somehow.

So, the observable pushes keeping the fact that we have a change in it, and the observers have to pull data.

There can be variations of this pattern, such as, instead of forcing the clients to pull data, you push data through the update() method that was called for each observer thorugh

notify(). This way, observers can get the new data they're interested in, then they don't need to have a reference to the concrete observable. In the terms of the UML diagram above, the arrow between concrete classes disappers.

So, there can have many different variations like a poll poll, or a push push, or a poll push kind of observer patterns.

This page have been intentionally left blank

DECORATOR PATTERN

Suppose you're an object, and you called the method speak(), which returns back a string "hello world".

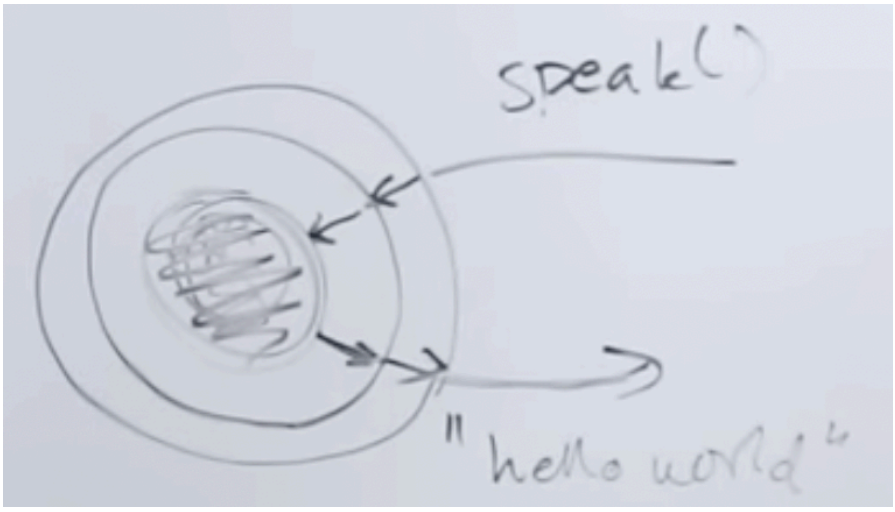
So, you send the method speak() and the object returns, it responds with a string.

Now, decorator pattern is a way of saying, if I want to change the thing that's returned, and if I want to change the behavior of this method, I could actually do that at runtime without changing the contents of this particular object. So, you change it at runtime and not at compile time. We can simply think this as changing the behaviour of this thing without actually rewriting the contents of this thing, without actuall yopening up the class and changing the contents of that file.

So, with decorator pattern, what we do is, wrap this object in another object, and when I send the message speak(), I am

sending it to the outer object which wil send it to the inner object, and we can do this indefinitely.

You can take another object and you wrap this wrapped object, and you wrap this wrapper in another wrappere, so another outer object on top of an outer object, that sends the message to the inner wrapper till it reaches the core.



But what's the point of making these wrapper kind of object? Well, that's the essence of decorator pattern. They are very valuable. Let's see.

The core object will be referred to as the component, and the outer wrappers will be referred as the decorators.

So, if we think about a single decorator here, it has a component and is a component as well, the case with just the core layer.

So, these decorators behave outside from the interface standpoint, they are of the same type as the core. In other words, they are exchangeable.

We wrap the original object, and they we can use the wrapper and pass that around as if it was one of these original things.

Let's look into the decorator pattern definition,

" The Decorator pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality. "

So, the outer circles are providing some additional features on top of the core functionaltiy. For example, toppings on the pizza. What kind of topping would you like?

But these additional features are fixated as such at compile time, they are optional and can be changed dynamically at run-time as per choice. How can we do this, let's find out.

But first let's discuss where this is useful? Like in a class explosion scenario, where you have too many subclasses, and it's all juggled up.

There's a saying, " Inheritance is not for sharing behaviours "

So, it's a way of using composition rather than inheritance in order to share behaviour.

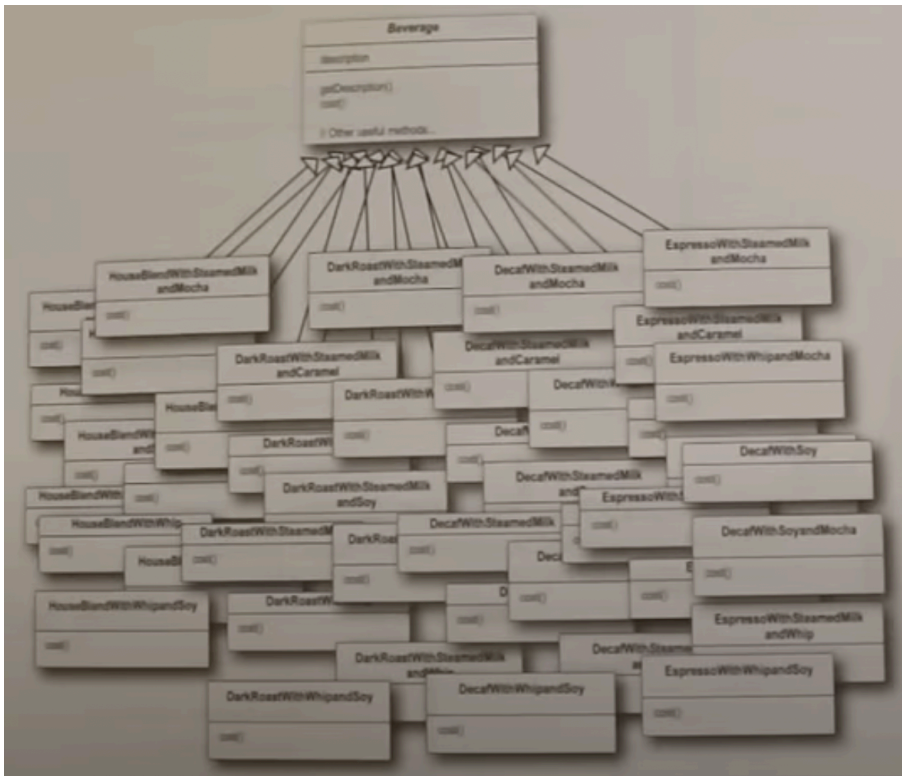
In our example, we're using inheritance just to make the outer wrappers as of the same type as the core, but it's not being used to share behaviour among them.

Suppose we have some subclasses A, B and C. We cannot compose a new class at runtime. But with decorators, what we're doing is we're defining the decorators A, B and C, then we can compose and use combination of those decorators at the same time, so it's more flexible.

For example, let's look at the example of a Coffee shop named Starbucks. It's about constructing a menu of products that can be ordered at the coffee house, so it's a bunch of different coffee types. So, a cusstomer comes in and asks for a particular type of coffee and they have tons and tons of variations of coffee.

Now, the coffee shop also provides flavours you can add on top of the type of coffee you choose. Now, just imagine that any flavor could be combined with any other flavor on top of the type of coffee. Starbucks has crazy combinations of the toppings or the extras that they have. And if you introduce a single class for each of these combinations on top of the type of coffee, you'll have what is commonly referred to as " class explosion "

Having tons and tons of classes each doing very very small things, from single point of responsibility that's okay, but it's completely unmanageable because you have so many of them.



Instead of this, what we can do is have the type of coffee classes as the baseline core class, and different classes that are acting as decorators, or we can say wrappers, on top of the baseline class.

Now, these decorators when used on top of the baseline classes are going to return the same baseline class type, with few variations about its behaviour.

So, the decorators refers to the thing it's wrapping, and in this way, the thing it's wrapping happens to be another decorator too if we have multiple decorators.

So, it recursively goes inwards to the core and recursively comes outwards to return the reference of the core with changed behaviour as per the decorations.

This behaviour is only possible when both the core and the decorators implements from the same kind of interface.

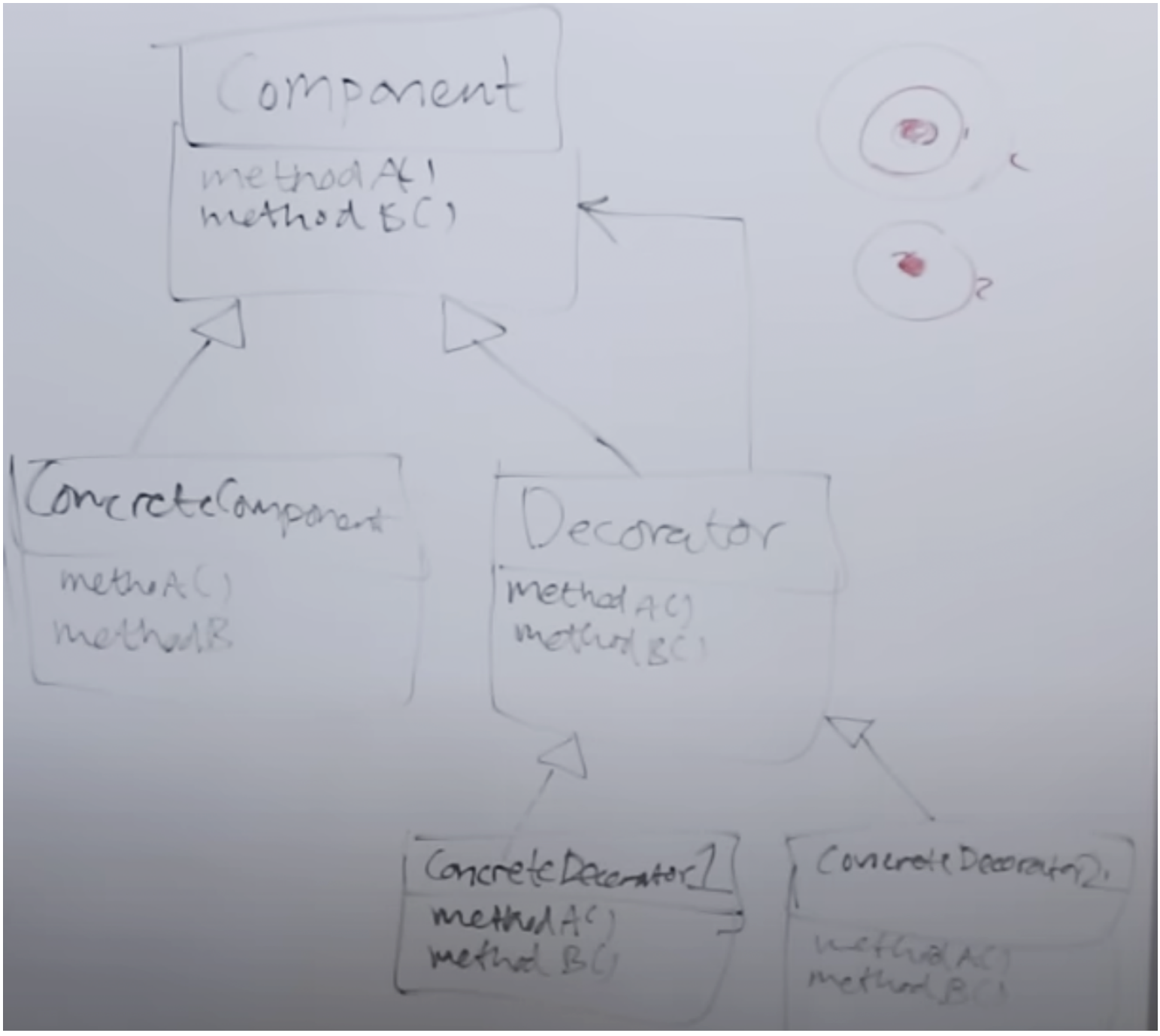
Let's look at the generalized UML diagram of decorator class,

We have a Component interface, which can also be abstract if we want default implementaitons. This component has been inherited by ConcreteComponent which actually has a implementation, and we can think of this as the core component. On the other hand, we have Decorator interface which implements the Conponent interface.

The Decorator interface not only is a component, it also has a component. It means that it itself is a component but it has another component. When we instantiate the decorator, we provide it a component. So, the decorator behaves as a component but also has a reference to another component, which can be also be another decorator because a decorator is in fact a component.

The Decorator is in fact just an interface, so it has to have ConcreteDecorator which implements from the Decorator interface. Now, we can have different types of decorators which

can extend from the Decorator interface, and can be used on top of the ConcreteComponent.



Let's look at the code example now,

interface Pizza {
StringBuilder name();

```
}

public interface Decorator extends Pizza {
}

public class CheesePizza implements Pizza {
public StringBuilder name;
CheesePizza(){
name = new StringBuilder("Cheese Pizza");
}
public void update(StringBuilder withSpicyToppings) {
name.append(withSpicyToppings);
}
public StringBuilder name(){
return name;
}
}

public class OnionPizza implements Pizza {
public StringBuilder name;
OnionPizza(){
name = new StringBuilder("Onion Pizza");
}
public void update(StringBuilder withToppings) {
name.append(withToppings);
}
public StringBuilder name(){
return name;
```

```
}
}

public class SpicyTopping implements Decorator{
public StringBuilder name;
public SpicyTopping(Pizza pizza) {
name = pizza.name();
name.append(" With Spicy Topping");
}
public StringBuilder name(){
return name;
}
}

public class SweetTopping implements Decorator {
public StringBuilder name;
public SweetTopping(Pizza pizza) {
name = pizza.name();
name.append(" With Sweet Topping");
}
public StringBuilder name(){
return name;
}
}

public class Main {
public static void main(String[] args) {
Pizza spicyPizza = new SpicyTopping(new CheesePizza());
```

```
System.out.println(spicyPizza.name());
Pizza mixPizza = new SweetTopping(new SpicyTopping(new OnionPizza()));
System.out.println(mixPizza.name());
}
}
```

```
Cheese Pizza With Spicy Topping
Onion Pizza With Spicy Topping With Sweet Topping

Process finished with exit code 0
```

This page have been intentionally left blank

FACTORY PATTERN

There are three design patterns of the factory method in general,

1. Simple Factory
2. Factory Method
3. Abstract Factory

Why do we need factory method? And why do we need such a thing called a factory?

As we remind of the Strategy pattern, we assume that we already such independent behavioural classes through which we construct a new object, which is essentially dependency injection on the new object we are constructing with those behavioural classes, because we are assuming that it would exist, or would have been created in the past. So, here we are programming by wishful thinking, you can think of it as that you put yourself in a situation where you say, " What if I already had a thing that did this? ". That's the way of abstracing away the construction of an object away from the place that uses that particular thing.

But now, at some point in the program, the thing you're passing around has to be constructed. If you think about it very carefully about the keyword new. You have to new up your class, and you have to instantiate your classes at some point in the program

inevitably. Then we need to instantiate objects somewhere and the question is where do we instantiate them? If we think about it, that's the thing factory pattern in general is trying to address.

Factory pattern is trying to say, when you are about to instantiate, let's encapsulate that instantiation so that we can make it uniform across all places so that you can use the factory whenever you wan to instantiate and the factory is responsible for instantiating appropriately.

Now, it might seem a bit silly to create a wrapper around the keyword new because if the difference is saying you're substituting new with create() method on factory class, then it seems like it's one line for the other. But wait a minute, think about it from these two different perspectives, it's possible that the instantiation is actually very complex. In other words, it's possible that you might need computation to initiate some kind of business logic in order to determine what parameters you want to pass to this particular object instance you are going to create. On the other way, it's also about polymorphism, if you have a factory that wraps your construction and if that factory is an instance then you can swap that at runtime, you can swap that instance for an instance of another factory polymorphism. We'll get into that in detail later.

In simple words, we might have different business logic that determines what thing you want to instantiate, and that logic can be encapsulated, can be built into, coded into, what we

commonly refer to as a factory, so this factory is responsible for holding, for keeping the business logic of creation of particular shared type.

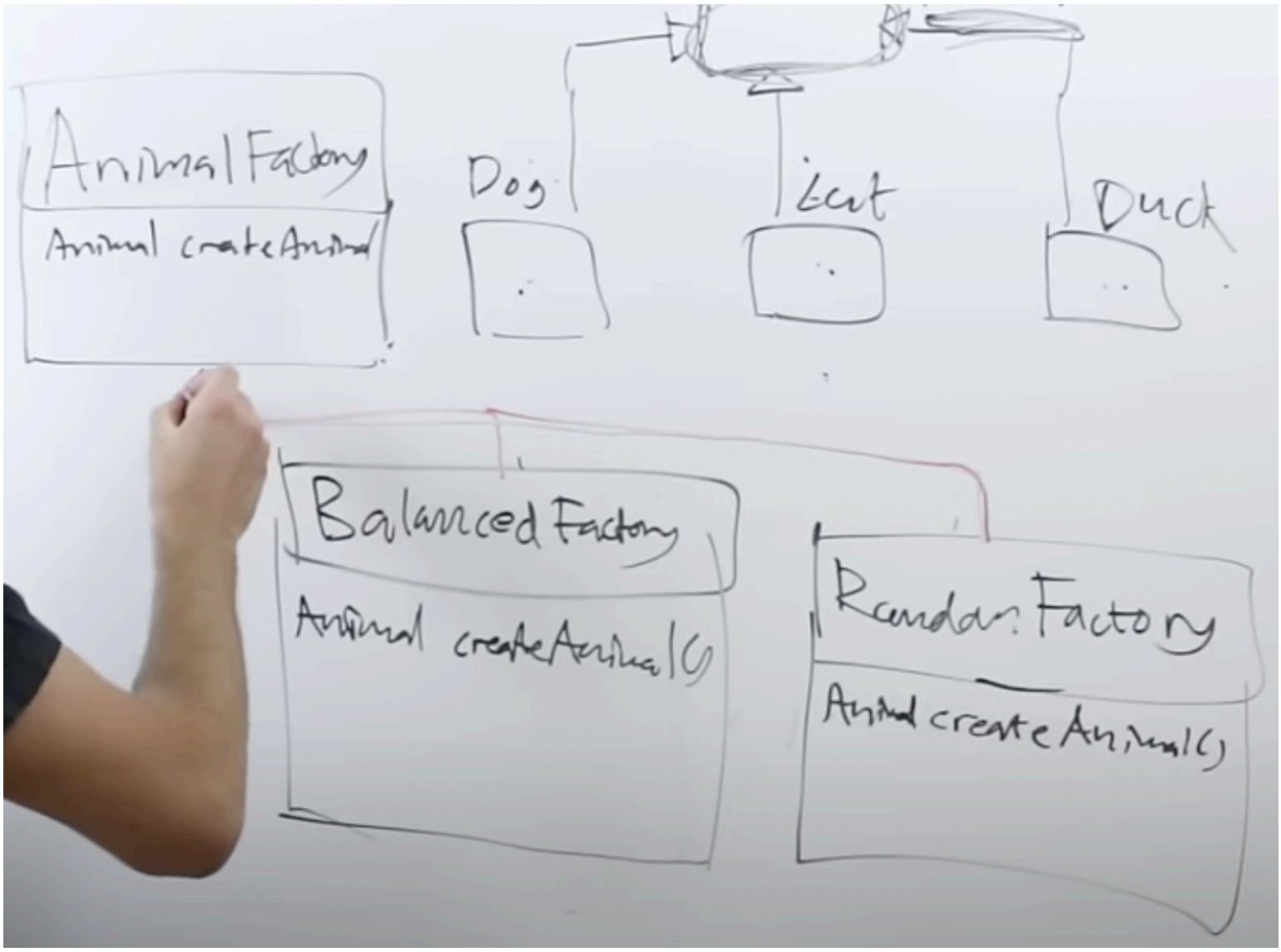
In general case or when we use a specific example, it would be a factory that's responsible for a particular creation mechanism, a particular way of constructing objects, so if you have two factories, then you have two ways of creating objects. They both create the same thing, but they create them in different ways.

Essentially, we have the things that we are creating, and we have the factories that create these things.

So, we're abstracting the logic of object instantiation from the user, in such a way that factory may handle it through a certain business logic, and return us the appropriate object as per the logic.

For example, We have RandomFactory which calls it's business logic createAnimal() method to instantiate any one of the Dog, Cat or Duck class from the Animal interface and return that object to the caller.

And suppose we have multiple factories that are instantiating objects in their own ways, in that case, we need to have a common interface between these factories in order to relate them as one concept.



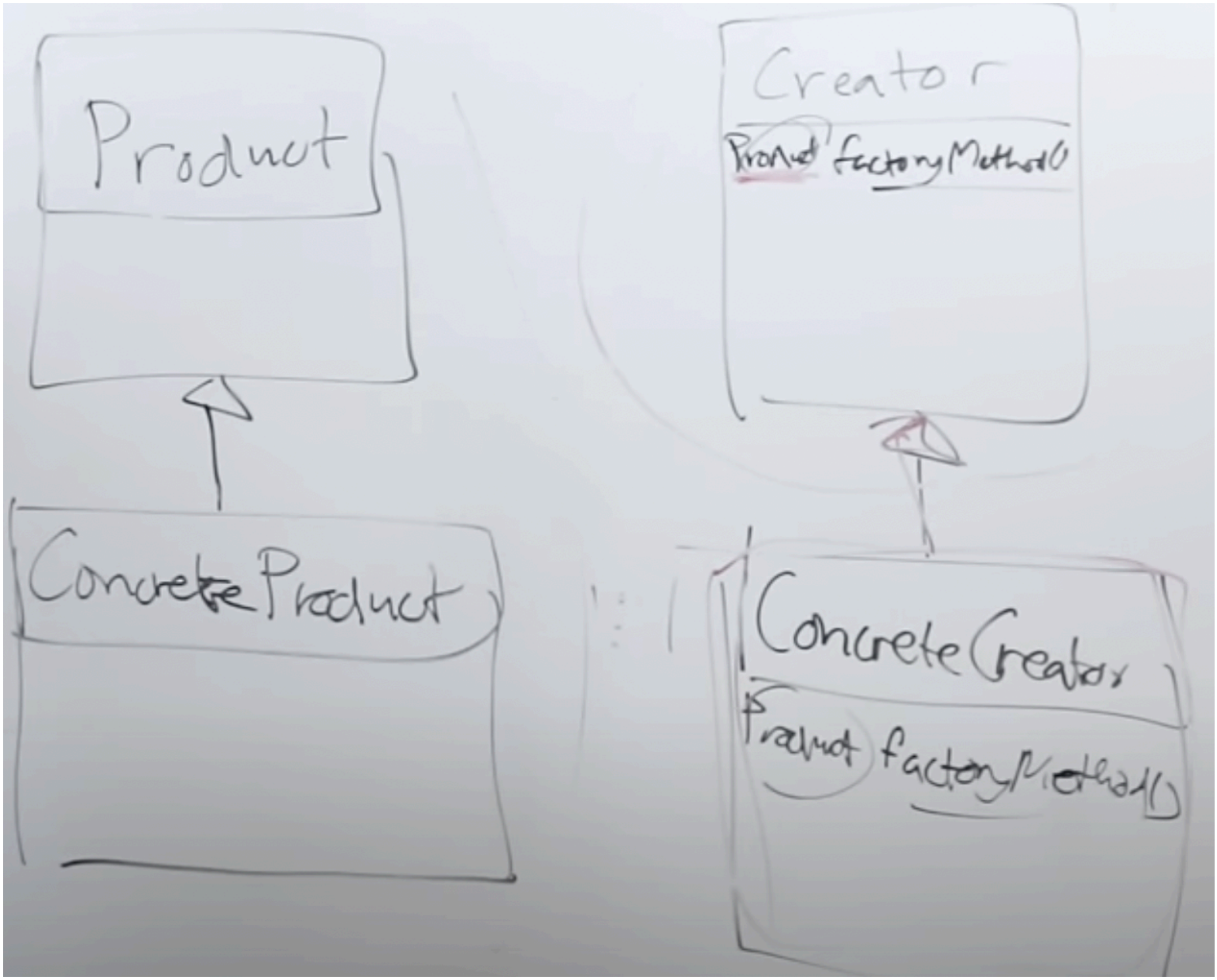
Okay, let's look into the actual defination now,

" The factory method pattern defines an interface for creating an object but lets subclasses decide which class to instantiate. "

When we're saying interface in this defination, it's in terms of a contract, it defines a common contract with under which you can refer to this particular thing. But the key for the factory pattern is that in the end, you want an obbject, you don't necessarily know how you want to construct that object, why you want to construct that object, and what parameters you want to pass when

constructing that object? These are all unknowns, and that's why you want to defer this, and let somebody else take that decision. And it's not only about which class to instantiate, it's also about what you actually want to pass to that class that you're instantiating.

Let's look into the UML diagram for factory pattern,



We have some kind of ConcreteCreator from the Creater interface, which are actually our factories, that creates some kind of ConcreteProduct that implements an Product interface, because whatever objects our factory create, they must have something common in them which is an interface.

So, creators are as the name implies concerned with creation of some particualr thing, of some shared type, which in this case is denoted as a product, so creators create products, but the creators might be abstract or it might be an interface.

Now, as we have heard about the simple factory as well, it simply means that having separate factories without any common creator interface between them, so it's essentially a way of having a single concrete factory, that is responsible for the creation of products.

Now, we can use a tiny bit of abstraction to gain much more flexibility, this notion of being able to create something of this shared type is an idea of which i could have multiple instances by some kind of a way of constructing a product. That is, Creator interface for multiple factories that implements it.

We have some kind of objects that we want to produce that we call producs, and how we produce them is defined by the factories that we call creators.

Factory method is a sort of a very helpful tool in trying to move away from a sort of class explosion into a world where we have a key key classes where there's a lot of variation that can happen by simplying combining them in appropriate ways and instantiating them with different properties so this is really the point about composition over inheritance.

For example, consider an asteroid game, where at each level, you have to spawn asteroid objects towards the spaceship, which we can do using factory method on asteroid objects, but suppose each level has it's own variation and difficulty level, and with so many kind of variations, how would we create so many types of factory methods for every level? For this purpose, we create factory based on their behaviours, like PowerUpFactory, SpeedUpFactory and so on, which separates the factory method implementation decoupled from the levels in which it is being used keeping flexibility.

Let's look at the code example now,

```
public interface Animal {
String name();
}

public interface Factory {
Animal create();
}
```

```
public class Cat implements Animal{
public String name(){
return "Cat";
}
}

public class Dog implements Animal{
public String name(){
return "Dog";
}
}

public class Duck implements Animal{
public String name(){
return "Duck";
}
}

import java.util.Random;

public class RandomFactory implements Factory{
public Animal create(){
int e = new Random().nextInt(3);
return switch (e) {
case 0 -> new Cat();
case 1 -> new Dog();
case 2 -> new Duck();
default -> null;
```

```
};
}
}

public class Main {
public static void main(String[] args) {
RandomFactory factory = new RandomFactory();
Animal firstAnimal = factory.create();
System.out.println(firstAnimal.name());
Animal secondAnimal = factory.create();
System.out.println(secondAnimal.name());
}
}

Cat
Duck

Process finished with exit code 0
```

This page have been intentionally left blank

SINGLETON PATTERN

Let's go straight into the definition,

" The singleton pattern ensures a class has only one instance and provides a global point of access to it. "

In other words, it's like you have a class and singleton pattern helps you to make it impossible to instantiate that class, except for a single time and whenever you want an instance, you will inevitably have to use that instance. The way it actually works, is that whenever you ask for an instance, you always get the same instance, so it's there already. Well, one hand is about making sure that you only ever have a single instance, and other hand is providing global access to that instance.

This seems pretty simple right, let's jump into the critiques to makes things interesting. There are two main points that argue about not using the singleton pattern.

One of the first point is that we have learnt to avoid globals, and to actually use scope. Imagine if everything is leaking into the global name space, it's extremely difficult to control because you have lots of lots of different things that might be ambiguous out of context or might be difficult to understand out of context, but probably more importantly, whenever you leak something to the global namespace, that thing might change without you knowing

it, and it's much harder to reason about your program, when you don't have control, and when you don't necessarily know the scope of a particular thing, because that thing interacting with whole program may get changed anywhere assuming it's changeable.

The second point about making sure that you only have a single instance that too is kind of an assumption you're assuming in the future, that I will only ever need a single instance of this particular class, and that isn't encessarily always true, especially if your application is growing, so you might need another instance someday.

For example, think about building a chat applicaiton, you might first think that the chat is a singleton, and you want to be able to reference the chat in which users are interacting. But then as time progresses, you start to realize that actually we're being very successful with this chat and we want to have multiple chat rooms, and if the single chat is managing users who are currently connected to the chat room, so it's just that when we were designing singleton, we didn't think about the fact that in the future, we maybe actually want multiple instances of this thing, that we currently believe to have a single instance of.

There's a great saying that " One man's constant is another man's variable. "

From another perspective, singleton will make testing very difficult, because whosoever is requesting singleton will definitely get a particular instance, but in testing you might want ot mock that, and that becomes a lot more difficult when you have this sort of global single instance that you're always reaching for. So, as soon as you testing, there's almost always a second needed instance because you might want to mock that within the testing of your application, so again, one man's constant may be another man's variable.

A better opinion to singleton pattern is that it's completely fine to have a single object only within your applicaiton, but it's not fine to force to make it impossible to create a second instance, because you don't necessarily know whether you at some point of time actually want the second instance.

So, here's how it works,

Let's take the same example of the chat room, so as per the Singeton's single responsibility principle, you're actually asking the chat room to now both be the global point of access to the one true instance of the chat room and you're asking it to behave as a chat room or you could even slice it as such in three divisons,

A. Behave as the global point of access to the single instance of the chat room
B. Ensure that there only ever exists a single chat room instance

C. To do whatever the chat room was originally intended to do

So, the interesting thing about Singleton pattern is that, we make the constructor of the singleton pattern private. We think that constructor has to be public so that you can construct instances of the obejct but you can actually make the constructor of a class private which means that from the outside, people can't construct that class, and that's the behaviour we're expecting in the singleton.

But hey, how can we have that single instance of the class, since constructor is now private, how we can create the single instance of that singleton whose constructor is private?

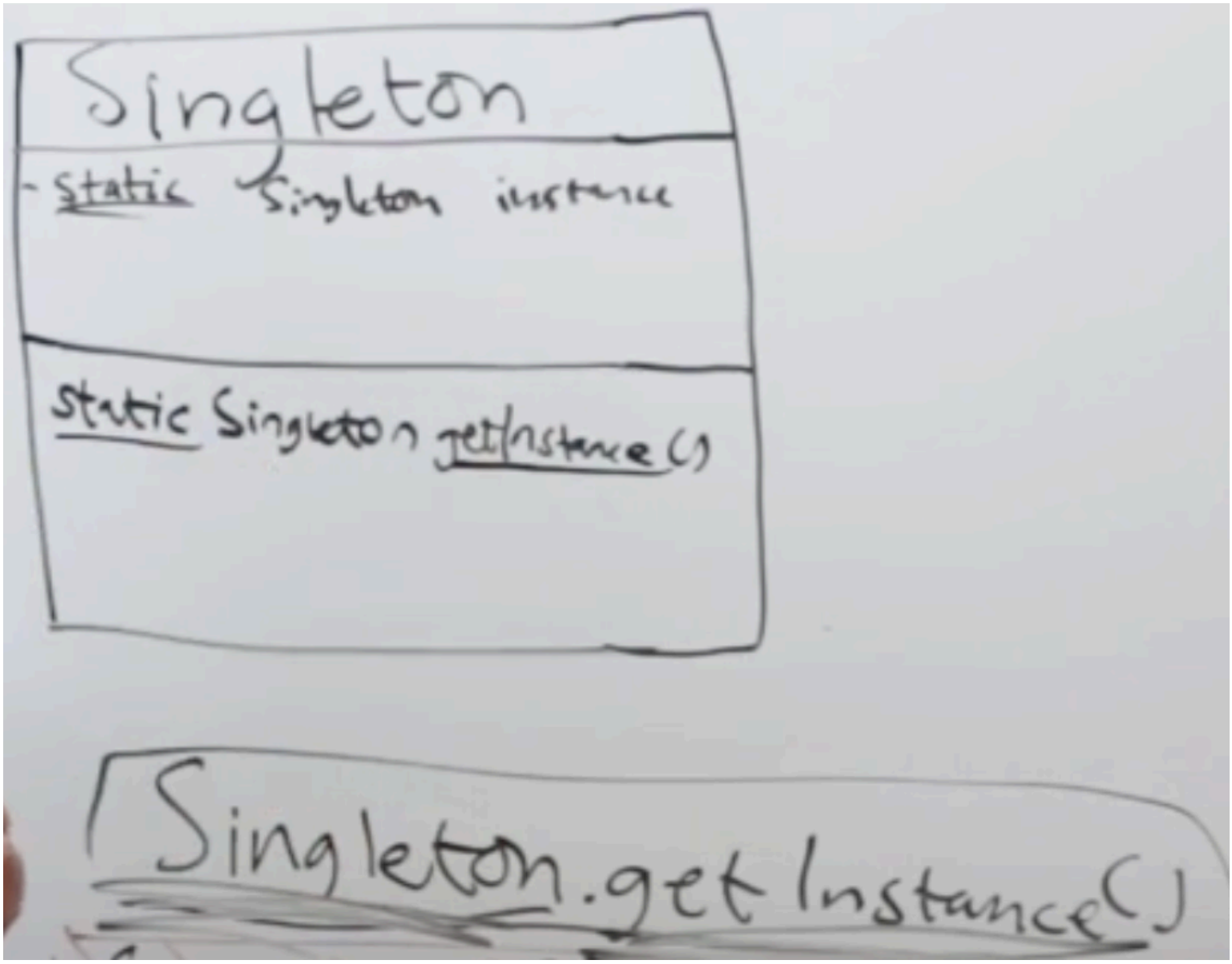
That's where static methods come into place, and some say that static methods shouldn't be used, but here, a static method is a class method, so it's still within the sort of the name space of the singleton.

A singleton has a static method called getInstance() and when you call the static method, which is essentially a class method, you get the instance of the singleton.

Here, note that getInstance() is a static method, it does not need us to use the new keyword with the constructor which in fact is private and will give compile time error.

The second thing is that singleton also has a static singleton instance, which ofcourse is a private variable, so that we couldn't globally access it without going through the getInstance() method.

So, the method getInstance() will check first, if there is an instance within this static variable called instance that holds a variable of type singleton, and if it does exist, it will return that instance, but if it doesn't exist, it will instantiate it just once, and save it into this variable, so that the next time we call getInstance() then there is an instance here.



Let’s look into the code then,

```
public class Singleton {
   private Singleton() {}
   private static Singleton instance;
   public static Singleton getInstance(){
      if(instance == null){
         instance = new Singleton();
      }
      return instance;
```

```
    }
}

public class Main {
    public static void main(String[] args) {
        Singleton s;
        s = Singleton.getInstance();
        System.out.println(s);
        s = Singleton.getInstance();
        System.out.println(s);
    }
}

singleton.Singleton@8efb846
singleton.Singleton@8efb846

Process finished with exit code 0
```

This page have been intentionally left blank

IN THE WORLD OF DESIGN PATTERNS, THERE ARE FOUR PATTERNS THAT ARE VERY MUCH ALIKE AND CONFUSING TO DIFFERENTIATE: **ADAPTER PATTERN, FACADE PATTERN, PROXY PATTERN, DECORATOR PATTERN**

Let's quickly go into what are the differences between these,

Adapter pattern is about making 2 interfaces compatible that aren't compatible.

Facade pattern is about taking a bunch of complex interactions and creating a facade that you can use instead of dealing with complex objects and compex interactions.

Proxy pattern is a way of placing a proxy between something you want to call so you want to call a particular thing, but instead of calling that thing, you call the proxy who calls that thing.

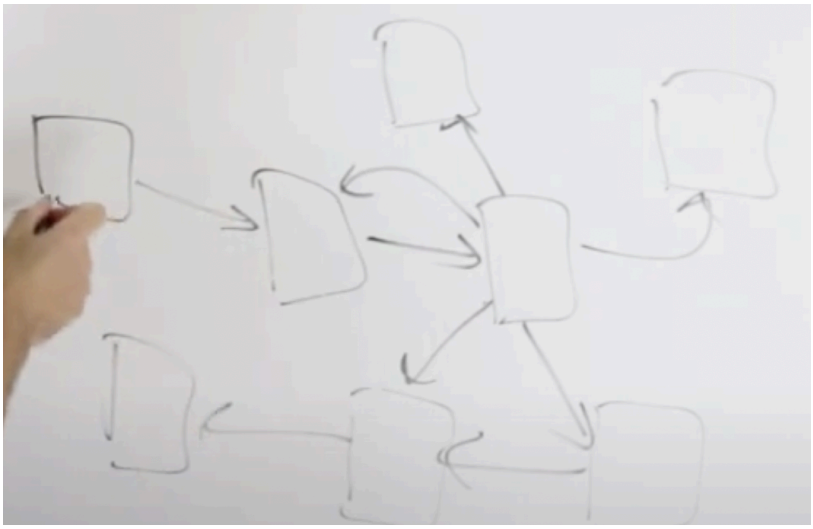
Decorator pattern is a way of adding behaviour to some particular object without actually opening that up and changing that object.

Again, adapter makes interfaces compatible, facade hides some complex logic, proxy intercepts a call and this controls access to the underlying object, And decorator adds behaviour to some particular object that it's decorating.

This page have been intentionally left blank

FACADE PATTERN

So, if you have got a bunch of things informally, say that these things represent classes, and then you have a bunch of interactions between these different things, and there's some level of compexity here, and then we've client as an end user from a particular piece of code using our piece of code, so this client wants to use your classes which are somehow loosely coupled with interactions between them, because we used single responsibility principle, so we might end up with using lots of classes in order to actually do anything meaningful.



For example, suppose you are building a compiler, you'd need some classes like a scanner, a parser, token symbol, and more, that you would probably use if you were building a compiler. In other words, it would make sense to separate things into these different classes, and have this high level of abstraction to separate to this extent. But that also means in order to do

anything useful with such a way, you might need to use lots of different classes at the same time which can actually get overwhelming.

So, we're saying dependning on your situation, we might have a lot of difficult wiring, to set up a data structure that you actually want in order to actually do any useful work.

Having sort of complex setup might be almost a natural consequence of having highly decoupled systems because you have two very general pieces that you compose together in order to do useful things but because they are so general they can be composed in a multitude of different ways which means that you can do a multitude of different things.

The facade pattern is a way of saying that it's fine to need to do all of this complex wiring but if I'm going to do this complex wiring then I want to have an easy way of doing it.

If you think about the word facade, the facade is the sort of exterior of the building, that faces onto the street or open space. So, it's hiding all of the abstractions within the building of how it's constructed and showing us the outside view of the building. So, it's like creating a nice wrapper, a nice interface that I can interact with.

So, we're saying that underneath the facade, we have a lot of plumbing, but outside of it, I don't want to have to deal with all of

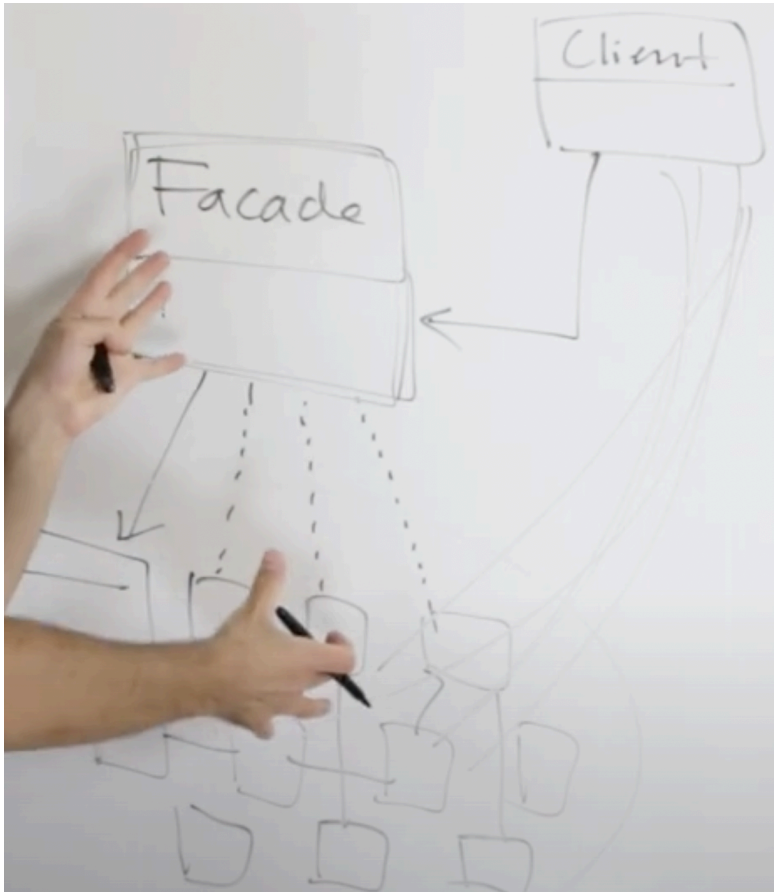
that plumbing, so we're creating this nice wrapper, nice interface I can interact with, instead of interacting with all the complex stuff and underneath wiring.

Let's look into the UML diagram,

So, a facade is the class that invokes methods on a bunch of other classes, which further invokes methods to other bunch of classes. The thing here is that the facade over that stuff isn't necessarily following any particular structure, and this stuff is completely dependent on your scenario.

The point is we construct a facade that interacts with these objects and simplifies the interaction for you. And we use a client class to interact with the facade instead of dealing the complexity ourselves. And there can ofcourse be multiple clients dealing with one piece of facade.

So, whenever you introduce a thing that you interact with instead of interacting with a bunch of other different complex pieces of classes, you're essentially using the facade pattern.



In facade pattern, we follow the " Principle of Least Knowledge "

In other words, the principle of knowing as little as possible, you can also think of it as talk only to your immediate friends, like objects should only talk to their immediate friends, and not the friends of their friends, because if you couple to only your friends, that's acceptable than coupling to even friends of friends.

Coupling in facade should be minimized because when we want to change things, it's very difficult to change because things are coupled. In other words, if I have this two things coupled to each other, in order to change one thing, I have to change the other as well.

For example, when the class representing me and the pen as couples, then if the pen suddenly changes in some behaviour, then I might have to change as well, because change is not compatible with the way I was expecting it to be.

There's a saying, " The golden grail is the low coupling and high cohesion "

So, these are attempts to make us reduce coupling.

Let's take a glance at the facade pattern definition,

" The facade pattern provides a unified interface to a set of interfaces in a subsystem, and defines a higher level interface that makes the subsystem easier to use. "

Let's look at the code example now,

```
public class Client {
public PlaceOrderFacade order;
}

public class PlaceOrderFacade {
PlaceOrderFacade(){
new OrderFromRestaurant();
new OrderPaymentGateway();
new UpdateOrderCatalog();
}
```

```
}

public class OrderFromRestaurant {
OrderFromRestaurant(){
System.out.println("Ordered At Restaurant");
}
}

public class OrderPaymentGateway {
OrderPaymentGateway(){
System.out.println("Order Payment Has Been Processed" );
}
}

public class UpdateOrderCatalog {
UpdateOrderCatalog(){
System.out.println("Order Has Been Placed");
}
}

public class Main {
public static void main(String[] args) {
new Client().order = new PlaceOrderFacade();
}
}
```

This page have been intentionally left blank

ADAPTER PATTERN

The adapter pattern is also known as a wrapper, because the adapter, what it does, is it wraps something so you can adapt to a specific interface.

In the adapter pattern, we have one type of input and one type of output. So, adapter maps from one interface to other interface.

So, there are some adapters that can take a variety of inputs but can give just one outlet as an output that we can plug in.

So, there are one to one adapters, and also many to many adapters.

So, we have an interface on one hand, and another interface on the other hand, and then you create an adapter so that the two things are compatible.

So, the exact definition of adapter pattern says,

" The adapter pattern coverts an interface of one class to another interface, and lets classes work together that couldn't otherwise because of incompatible interfaces. "

Here, in this definition, interfaces refers to some form of contract and not just an interface.

So, let's start with a client.

This client refers not a customer, but refers to some piece of code that wants to use something else. Client here is not the part of the pattern, but the user of the pattern.

The client uses something which is of type Target which is an interface, so the client cannot directly use an interface Target, it needs to use something that implements Target. So, this is the method in the interface that client expects.

So, as in a adapter, the client can only see and deal with the input, but cannot directly access the output.

So, in our interface Target, we take the method request() that client will call, as the signature of the thing that client is expecting, so that it behaves in a particular way in order to give the output as expected like in an adapter.

Now, who implements the Target interface, that is the Adapter class, which has to define the funciton request() of the interface.

Now, somehow we need the Adapter class to reformat the behaviour from input to the output. How do we do that?

So, this adapter is taking in an request call, but it delegates to, another class, something which follows a completely different contract, a completely different interface, and we call it Adaptee, because it's the thing being adapted.

The method in Adaptee class follows a completely different signature as that of the Adapter class, suppose it's specificRequest()

So, what we do is the client invokes the request() method from the interface to the Adpater class, which follows the signature that we are used to use it, that is the request(), and then we can call the Adpater class to call the Adaptee method specificRequest(), that means the Adapter has the Adaptee, delegating the request down to the Adaptee, and then the Adaptee can perform the specific request that we were originally interested in doing.

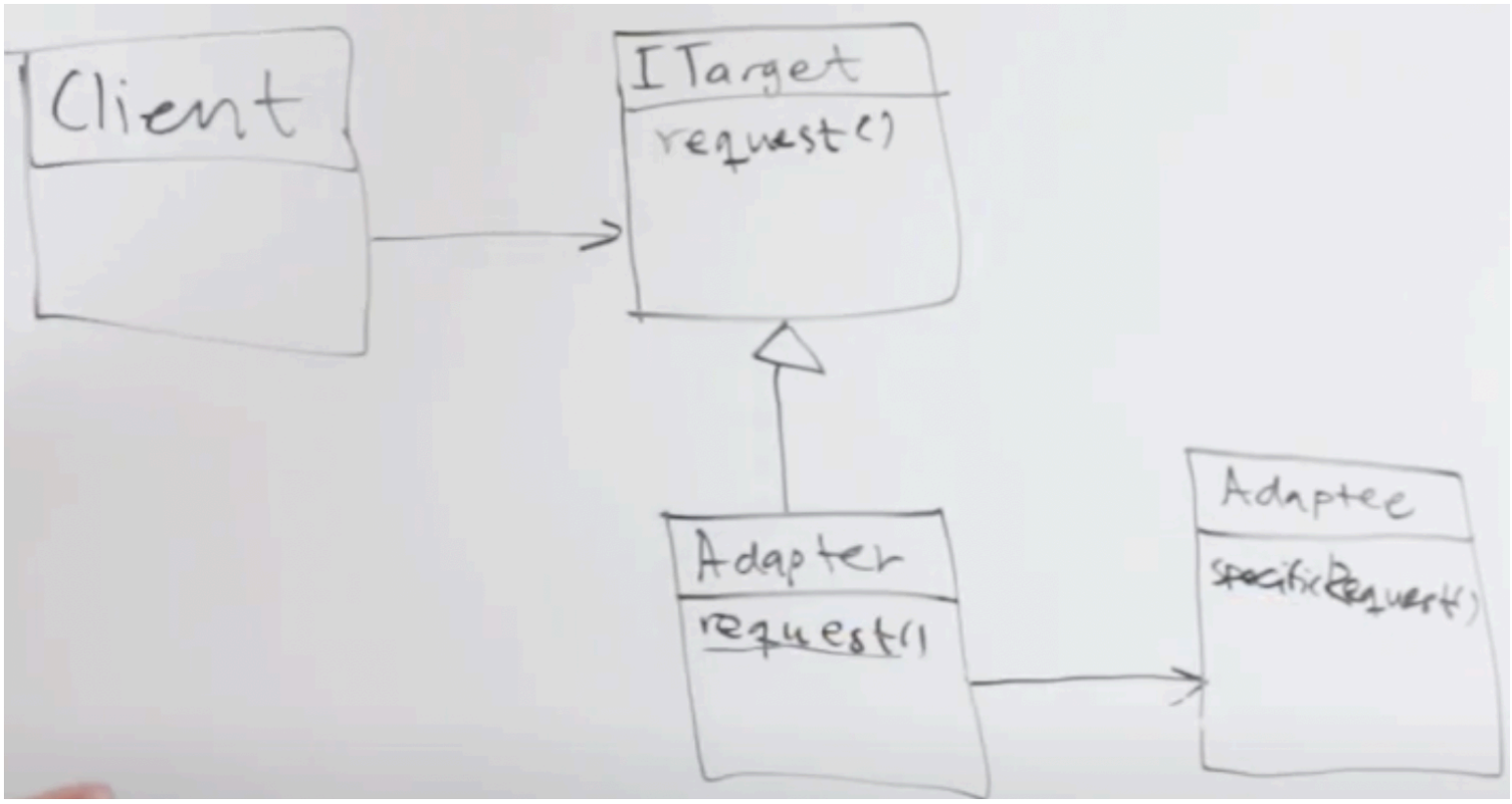
So, suppose that the standard interface using request() method is scattered across a lot of different places, and it'll difficult for us to change that to this specificRequest() style directly, or in the future, we might have to change the signature specificRequest() in the Adaptee, so we're encapsulating that particular change for us to make it easy to change it in the future.

So, what are doing is putting an adapter between the client and the adaptee, so that if we change our minds about what we want

to adapt as an adaptee in the future, we kind of can exchange that pretty easily.

In a nutshell, the client wants to call specificRequest() but only by using the signature of the interface request().

So, the intention is to not change the underlying behaviour to adapt something.



So, let's look at the code.

```
public interface Target {
void request();
}
```

```
public class Adaptee {
public void specificRequest(){
System.out.println("Request accepted by Adaptee");
}
}

public class Adapter implements Target {
Adaptee e;
Adapter(Adaptee e){
this.e = e;
}
public void request() {
System.out.println("Request passed to Adaptee");
e.specificRequest();
}
}

public class Main {
public static void main(String[] args) {
Target e = new Adapter(new Adaptee());
e.request();
}
}
```

This page have been intentionally left blank

COMMAND PATTERN

Let's get straight into the actual defiinition.

" The command pattern encapsulates a request as an object thereby letting you parameterize other objects with different requests queue or log requests and support undoable operations. "

What does it say, huh? Let's understand it.

Okay, so you've got an object and somebody sends a request to that object for something or to do something and the point here is that request is what we're trying to encapsulate, so we're not about to encapsulate the object sending the request, we're about to encapsulate the actual request going between. So, we're actually encapsulating a command, we're not about to encapsulate somebody receiving a command nor somebody sending a command but rather than actual command itself.

So, we can then take a bunch of these encapsulated commands and compute them into some kind of context so again we have the receiver and we have the sender and we encapsulate the command into some kind of object, and if you have multiple commands as obejcts, you can suddenly pass these objects around.

So, we have now things that are sets of comamnds so you have a lsit of commands that you're about to execute or you can have a collection of commands that you execute when you press different buttons, and then you invoke the commands one by one, so then you have a queue of commands that you execute, or with a composite pattern, you can even make like a batch command, like creating a command that itself contains multiple commands.

So, you have a command that does something, the inverse of that command, is an undoing right. So, if I have a command that adds something, and then I have the inverse of that commad that substracts that something, so if you have a bunch of commands that are encapsulated and all of them follows this logic being undoing as well, that's the demand of the command pattern.

So, if you have a queue of commands that you execute in a aeries you can then undo backwards because if you undo them in the same opposite order of which of the order you executed them then you would end up at the start.

Let's take the example of philips hue light that can be controlled through remote and smartphone app connected to wifi.

Let's say you're running a company and you are tasked with building the smartphone app that users like me use to program this device. In other words, when I press on, I might want a different thing to happen then when you press on in your device.

So, when we're building such kind of a device, the word ' dependency injection ' should really come to mind, because we're saying that the action sof any particular button here can't be hardcoded into this invoker, so this invoker invokes commands right, we can't hardcode commands on these particular buttons because we want users to be able to design that themselves, so this is why we want to have the notion of a command, a set of commands.

The same architecture can actually be used regardless of which thing I'm plugging in, because the one thing that's the sam eacross all of the things that we're talking about is that we want to press a button, wna tot send a command to a particular device, that does a particular thing, and this whole notion about attaching the command to the button and then sending the command whenever we press the button. That's completely the same thing across the board. We construct a command, we inject that command into the invoker, and whenever the invoker is pressed then we send that command off and execute that command, and that command might do something to something else, and that something else is a receiver.

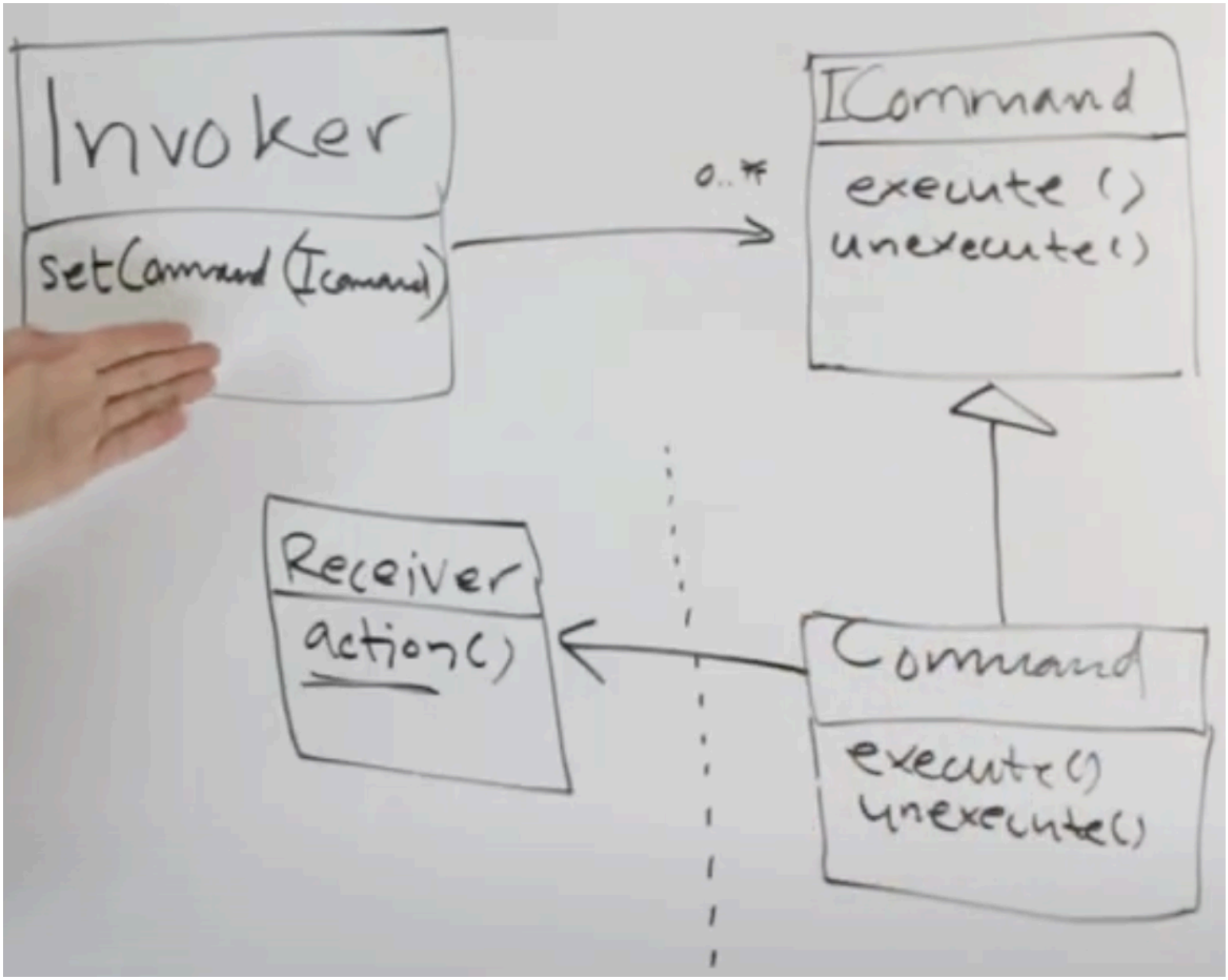
So, we attach a command the invoker, and the command when it's executed actually does something to a receiver. You can have a lot of invokers that are just coupled to the comamnds, and any command can do anything it wants to any particular

receiver. If you're sharing interfaces with receivers, then you can even use the same receiver for multiple commands.

Let's look at the UML diagram now,

So, we have an Invoker, and the invoker has one to many iCommand interfaces, which are not concrete so the invoker couples to the interface. The iCommand interface has a number of implementations and these we call Command class. And the Command class has a Receiver class.

Now, the methods are, that the invoker has a setCommand() method that takes a iCommand as an argument. The iCommand interfaces takes two methods, one is execute() and the other is unexecute(), so the execute() does the thing and the unexecute() undoes that thing. The conrete Command class defines these two methods from the iCommand interface. Finally, the execute() method in the concrete Command class will probably invoke a method action() which performs some action on the receiver.



We're saying that the Invoker has many iCommands, and the particular iCommand is a Command and any particular Command can act upon a Receiver.

So, to put it simply, the invoker acts on the receiver.

Let's look at the code now,

```
public interface iCommand {
void execute();
```

```
void unexecute();
void getReceiver();
}

public class Invoker implements iCommand {
public iCommand c;
public void setCommand(iCommand c){
this.c = c;
}
public void execute() {
c.execute();
}
public void unexecute() {
c.unexecute();
}
public void getReceiver() {
c.getReceiver();
}
}

public class ColorCommand implements iCommand{
public Light l;
ColorCommand(){
l = l.getInstance();
}
public void execute() {
l.colorYellow();
}
```

```
public void unexecute() {
l.colorWhite();
}
public void getReceiver() {
l.getLight();
}
}

public class LightCommand implements iCommand {
public Light l;
LightCommand(){
l = l.getInstance();
}
public void execute() {
l.lightOn();
}
public void unexecute() {
l.lightOff();
}
public void getReceiver(){
l.getLight();
}
}

public class Light {
public boolean state;
public String color;
private Light(){}
```

```
private static Light l;
public static Light getInstance(){
if(l == null){
l = new Light();
}
return l;
}
public void lightOn(){
state = true;
}
public void lightOff(){
state = false;
}
public void colorWhite(){
color = "White";
}
public void colorYellow(){
color = "Yellow";
}
public void getLight(){
System.out.println("Color is " + color );
System.out.println("Light is " + ((state) ? "Turned On" : "Turned
Off") );
}
}

public class Main {
   public static void main(String[] args) {
```

```
    Invoker invoke = new Invoker();

    invoke.setCommand(new LightCommand());

    invoke.execute();
    invoke.getReceiver();

    invoke.setCommand(new ColorCommand());

    invoke.execute();
    invoke.getReceiver();

  }
}
```

```
Color is null
Light is Turned On
Color is Yellow
Light is Turned On

Process finished with exit code 0
```

This page have been intentionally left blank

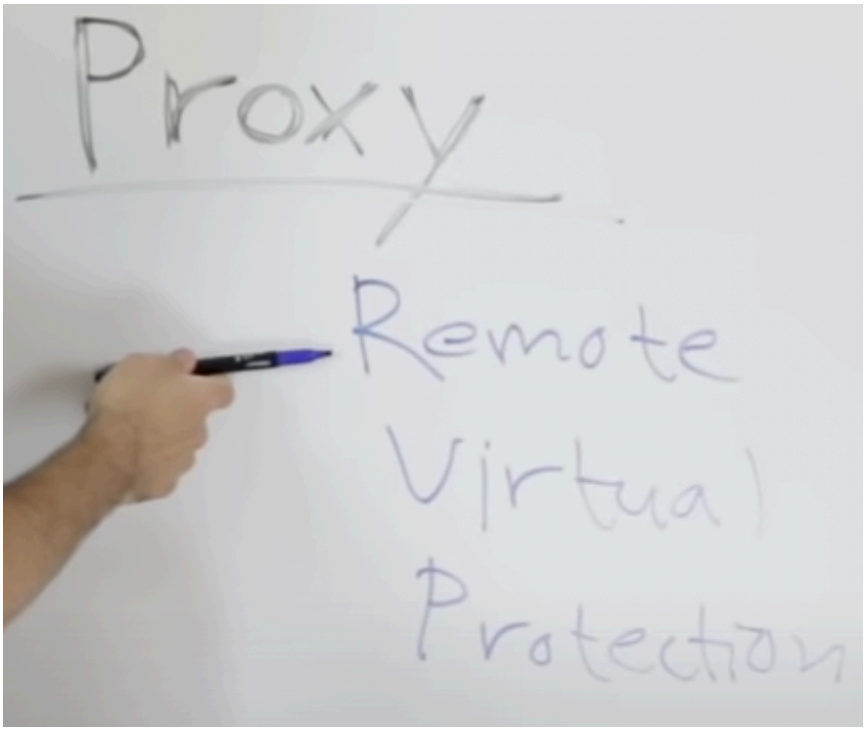
PROXY PATTERN

" The proxy pattern provides a placeholder for another object to control access to it. "

So, this is just a way of saying that, you have an object that you want to get access to that, and you want to invoke methods on that, and be able to somehow interact with that, but instead of being allowed to interact wiht that thing, you interact with a proxy who interacts with that thing, so it's another way of introducing another level of indireciton, instead of calling the thing you wan to call, you call something else that calls the thing you want.

It's not only about how you string the objects together, it's also about the intent like some patterns are mostly different in intent, rather than in the technicalities, so they might be doing on the surface or technically exactly the same thing, but what's different is that you are doing it for a different reason. So, the word proxy means to communicate to other developers. With proxies, we're trying to solve a specific set of problems that are all access related. So, you put a proxy in front of something that you want to allow people to access but you have the proxy so that you can control access to that thing.

A proxy can either be a remote proxy, a virtual proxy, or a protection proxy.



Now, a remote proxy is suggessted to be used when you want to access a resource that's remote. What do we mean with remote here, is that something that could exist for example in a different server, or in a completely different namespace. So, you have to leave like the safe boundary of your application out into the outside world, in order to retrieve some information back, so this sort of interaction or transaction could be wrapped in a proxy, so your proxy is responsible for interacting with the remote resource and giving you back the things that you need.

If we're familiar with promises in javascript, we can think of the proxy as something that would interact with remote resource but immediately return you a promise. So, it returns you a promise that promises to evaluate to the concrete resource that oyu were looking for after a while but instead of scattering this http request

or socket connection, you have this proxy that simply interacts with this remote resource on your behalf.

So, one of the key things of the proxy pattern is that it looks like the remote resource, it looks like the thing it's proxying. So, if you think about adapter pattern for example, adapter pattern adapts to a different interface if you want to access a thing that has some particular interface but you have a different interface and you want to acess it via your different interface. Proxy pattern is simply a way of controlling the access so it doesn't change the interface. So, when you have something that you want to interact with, that particualr interface, you simply want to intercept the accessing of that thing that you want to access for some reason such as security or caching,

Now, onto virtual proxy, a virtual proxy controls access to a resource that is expensive to create. So, this is like caching. You got some object which you want to interact with but you know that creating that object is expensive so you put something in front that you can interact with instead and then that proxy makes sure that only when you really need it do then do interact with the actual underlying thing.

A protection proxy, we can think about as access management. So, you've got a user and you're not sure whether a particular user has the rights to access a particular resource so you stick this protection proxy in between and that protection makes sure that only users that are allowed to access the underlying

resource do get access to the underlying resource. Generally, the point is that it controls acces to a resource based on access rights. So, you make a call to some underlying thing and the question is whether you are allowed to make that particular call. The proxy here makes sure that if you aren't allowed to make that call, then you just can't make that call.

Essentially, proxy adds some additional behaviour not in the sense of decorator pattern but in the sense that it adds additional behaviour with the intent of controlling access to the underlying object.

Let's take a example of a book parser to take a deep dive into one of the proxy patterns, virtual proxy.

We are taking a string representation of a book in some text format, and the point is that the string is huge. And we have a class named as the BookParser that we can instantiate with the string containing the book information. We can then access book information from this book parser class that has that book string processed, like numberofPages(), numberOfChapters() and so on.

Suddenly we realize that we have a performance problem we need to for some reason, because at the time of object instantiation of BookParser, the processing the book string takes a while, but it is of no use if information is not retrieved by the client using the functions in that BookParser. So, what we're

going to do is add a proxy in between so we know that the call to the book parser is an expensive one and the call to these methods of the book parser are cheap. And we also know that the client won't always call this second method, and may leave it after instantiation. So, as per this hypothesis, we can improve performance in this scenario by constructing the expensive operation to only happen when if the client actually calls the second cheap method. So, we want to defer, and we want to make it lazy, and that's why we want to sstick a proxy in between.

So, to put simply, instead of the client directly interacting with the book parser, the client now interacts with the proxy who interacts with the book parser. The client instantiates the proxy, but the proxy doesn't instantiate a book parser unless the second cheap method is called by the client. But the idea here is that the proxy has the same interface as the one it's a proxy of, so the client can keep behaving and can keep passing around this proxy as if it was a book parser in the same way, but internally it's being deferred.

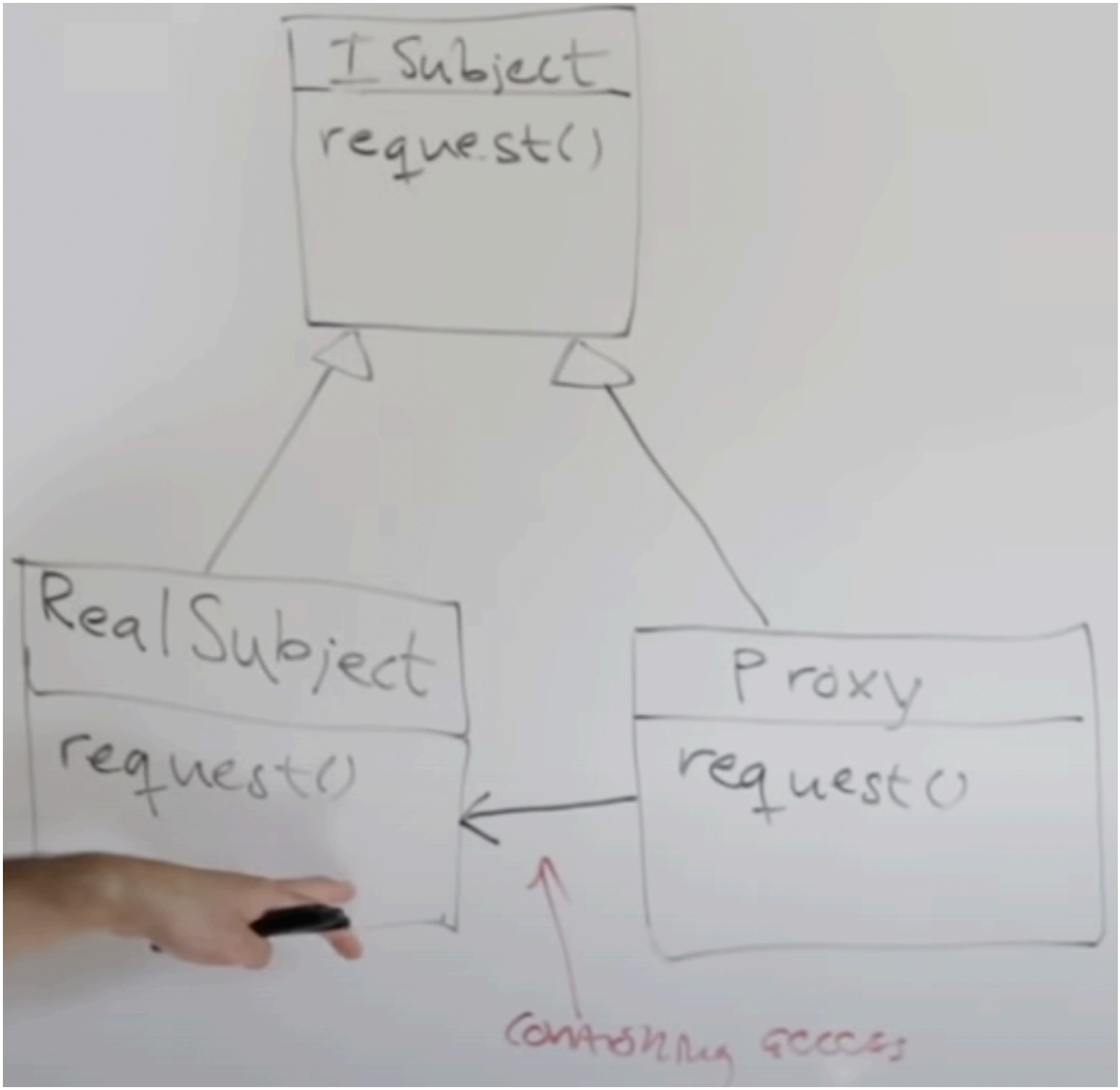
Let's discuss UML diagram of proxy pattern.

So, we have an iSubject interface that has a request() method which has to be defined further in classes that implements it.

Now, most of the times when we have a realization of an interface, we've been calling it concrete, but here instead of

calling it ConcreteSubject, we'll call it RealSubject. The reason is that it emphasizes that this is the thing we had before we introducted the proxy or this the thing that contains the underlying actual thing we want to do but then we stick a proxy in between and the proxy will also be concrete so we have two things that are concrete, and that's the reason it has been named RealSubject.

And because the proxy also behaves as the subject, it'll also implement the iSubject interface, so that's how you get the substitutability. But here what is important is that the Proxy class has a RealSubject class, which means that it can delegate to the RealSubject, as well as instantiate it. Because that instantiation is actually the expensive operation that we wanted to control access to, so we're creating a proxy for it.



The general point is that proxy controls the access to the real subject, and proxy follows the same interface as the thing it is proxying making it interchangeable with the thing it's proxying, and the proxy doesn't necessarily have to be passed to the real subject upon the instantiation.

Let's look at the code.

```
public interface iBook {
   void getBookInfo();
   void getAuthorInfo();
}

public class RealParser implements iBook{
   private final String bookInfo;
   private final String authorInfo;
   RealParser(String book){
      bookInfo = book.split(",")[0];
      authorInfo = book.split(",")[1];
      System.out.println( "Book has been parsed ");
   }
   public void getBookInfo(){
      System.out.println(bookInfo);
   }
   public void getAuthorInfo() {
      System.out.println(authorInfo);
   }
}

public class LazyParser implements iBook{
   private RealParser realParser;
   private final String book;
   LazyParser(String book){
      this.book = book;
```

```
    }
    private void firstParse(){
        if(realParser == null){
            realParser = new RealParser(book);
        }
    }
    public void getBookInfo() {
       firstParse();
       realParser.getBookInfo();
    }
    public void getAuthorInfo() {
        firstParse();
        realParser.getAuthorInfo();
    }
}

public class Main {
    public static void main(String[] args) {
        String book = "BookInfo: The_Book_Of_Rose, AuthorInfo:
Aryan";
        var bookParser = new LazyParser(book);
        bookParser.getAuthorInfo();
    }
}

Book has been parsed
 AuthorInfo: Aryan
```

This page have been intentionally left blank

ABSTRACT FACTORY PATTERN

An abstract factory is a set of factory methods. So, an abstract factory makes use of multiple factory methods. So, let's first see the definition of the abstract factory pattern.

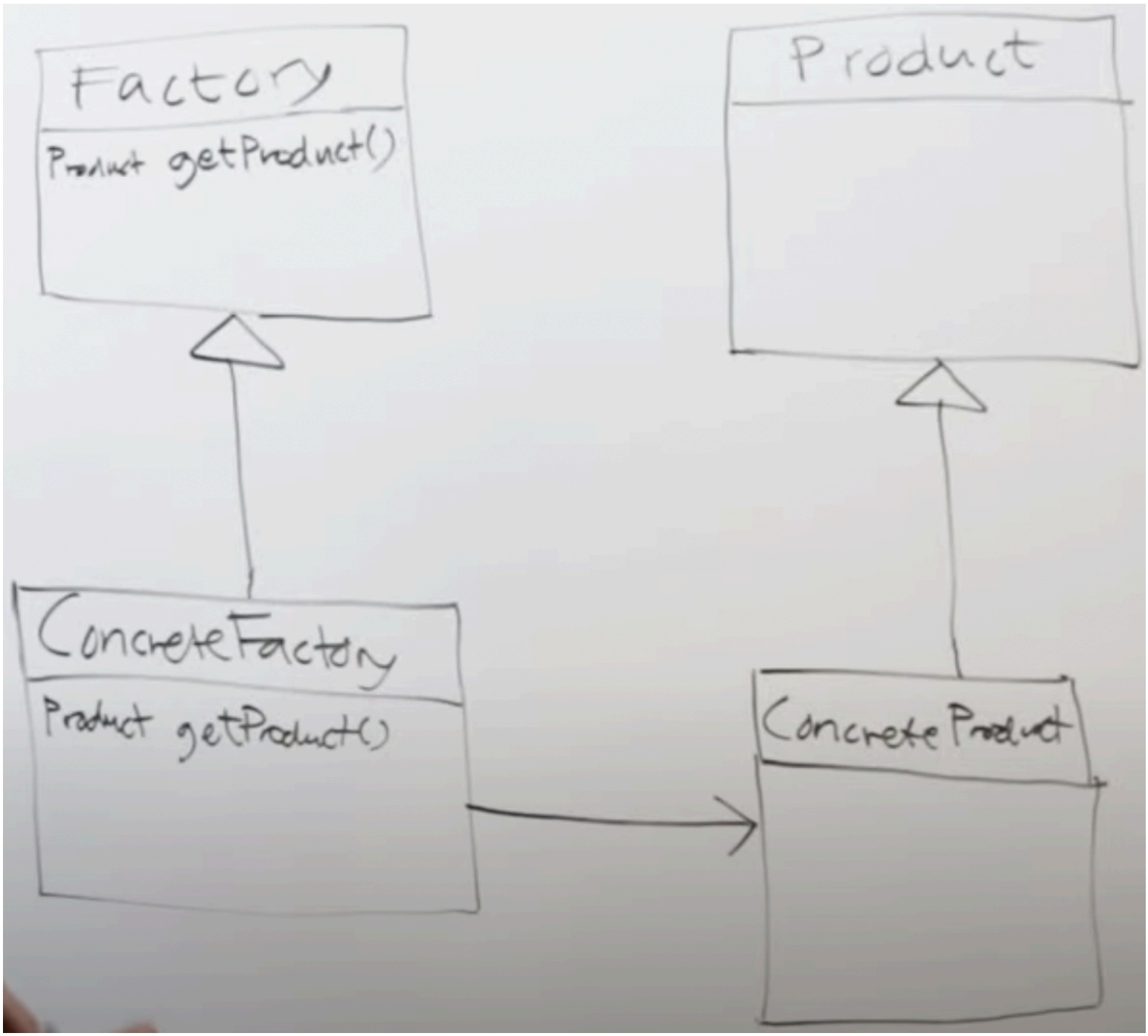
" The abstract factory pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes. "

The single difference between the factory and abstract factory is that factory method constructs a single object whereas the abstract factory method constructs multiple objects.

Let's recap the UML diagram of the factory method pattern.

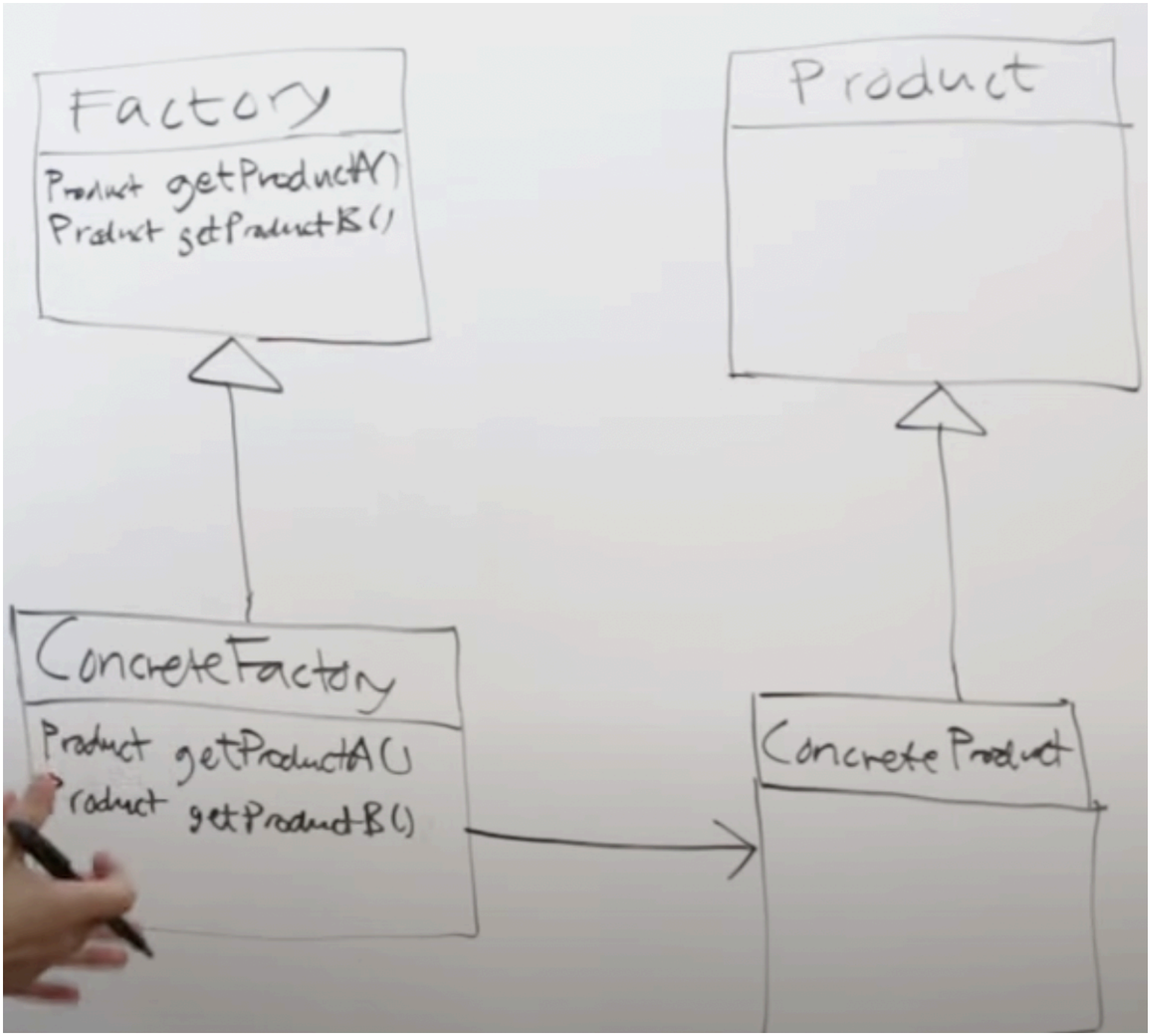
In factory method, we had a factory interface or an abstracat class, which has a number of concretions that are concrete factories. The factory has a factory method that returns some kind of product. But what is this product that we are talking about? So, we need to have a class that's a product. In a nutshell, the factory creates products. The factory is an interface of which there are multiple implementations and any such concrete factory will create a product if you call the factory method getProduct(). And because product is an interface or an abstract class, there can be numerous implementations of the product. So, we need to have a concrete product.

And lastly, we show that the concrete factory creates the concrete product. In the general plane, there are factories that create products, and depending on which particular concrete factory you instantiate, it might create different concrete products, with different kinds of logic and parameters.



Now, what is the difference between this and the abstract factory? Here's the simple change we need to make.

The difference is that any factory does not only have the getProduct() method, but let's say we have two methods named getProductA() and getProductB(). Now, we are saying that any factory does not only create a single product but has the capability of using multiple products. So, when you have a concrete factory, that concrete factory cannot simply just have implementation of a single product, but it can have implementation of multiple products.



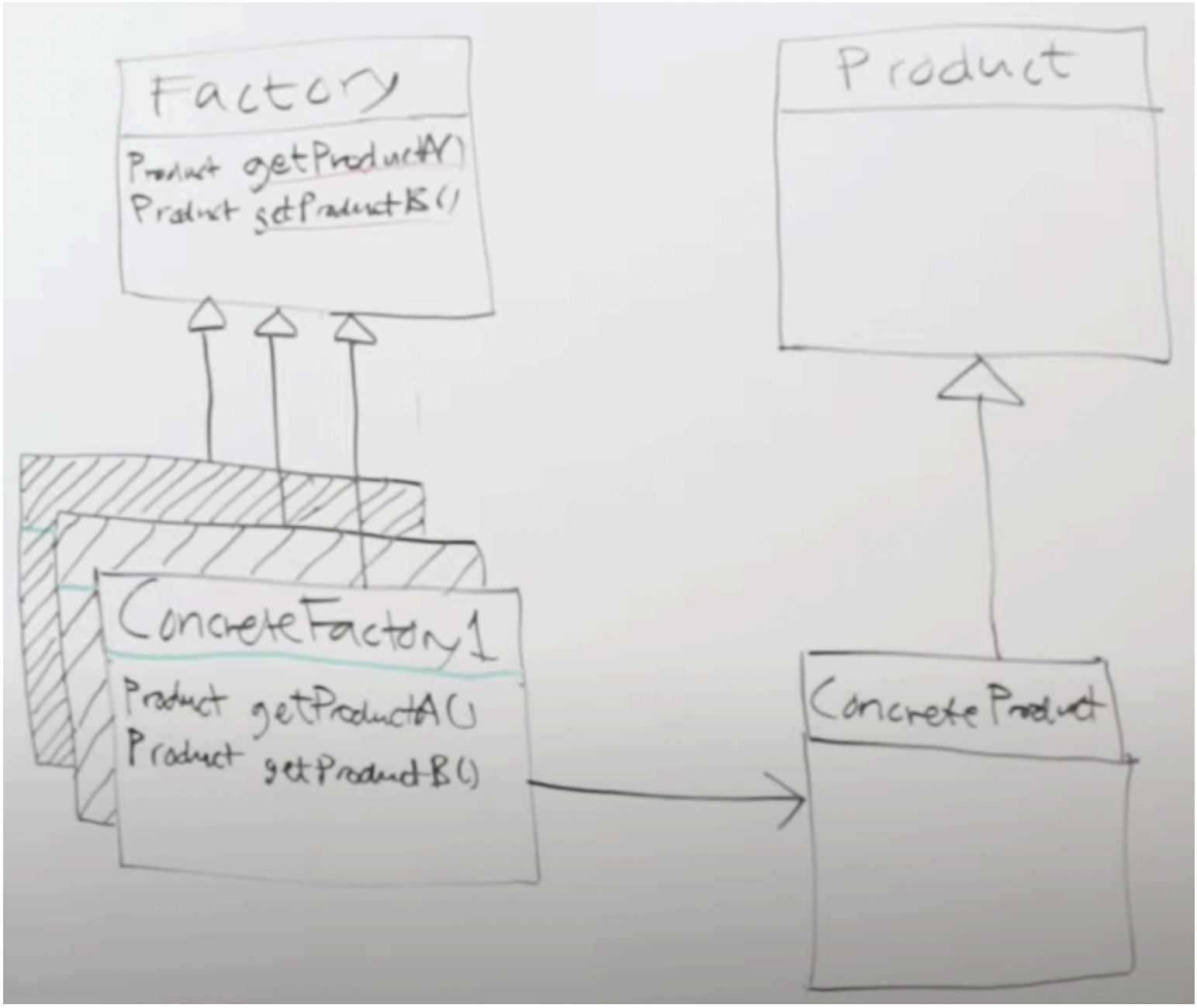
So, as per the abstract factory definition, we have created a family of related objects, that is getProductA() and getProductB(), without specifying implementation of this product, in other words, whether it's one concrete product or another concrete product is not a responsibility for the factory.

A common example for where abstract factory pattern is useful is when you are building UI controls that need to be platform independent.

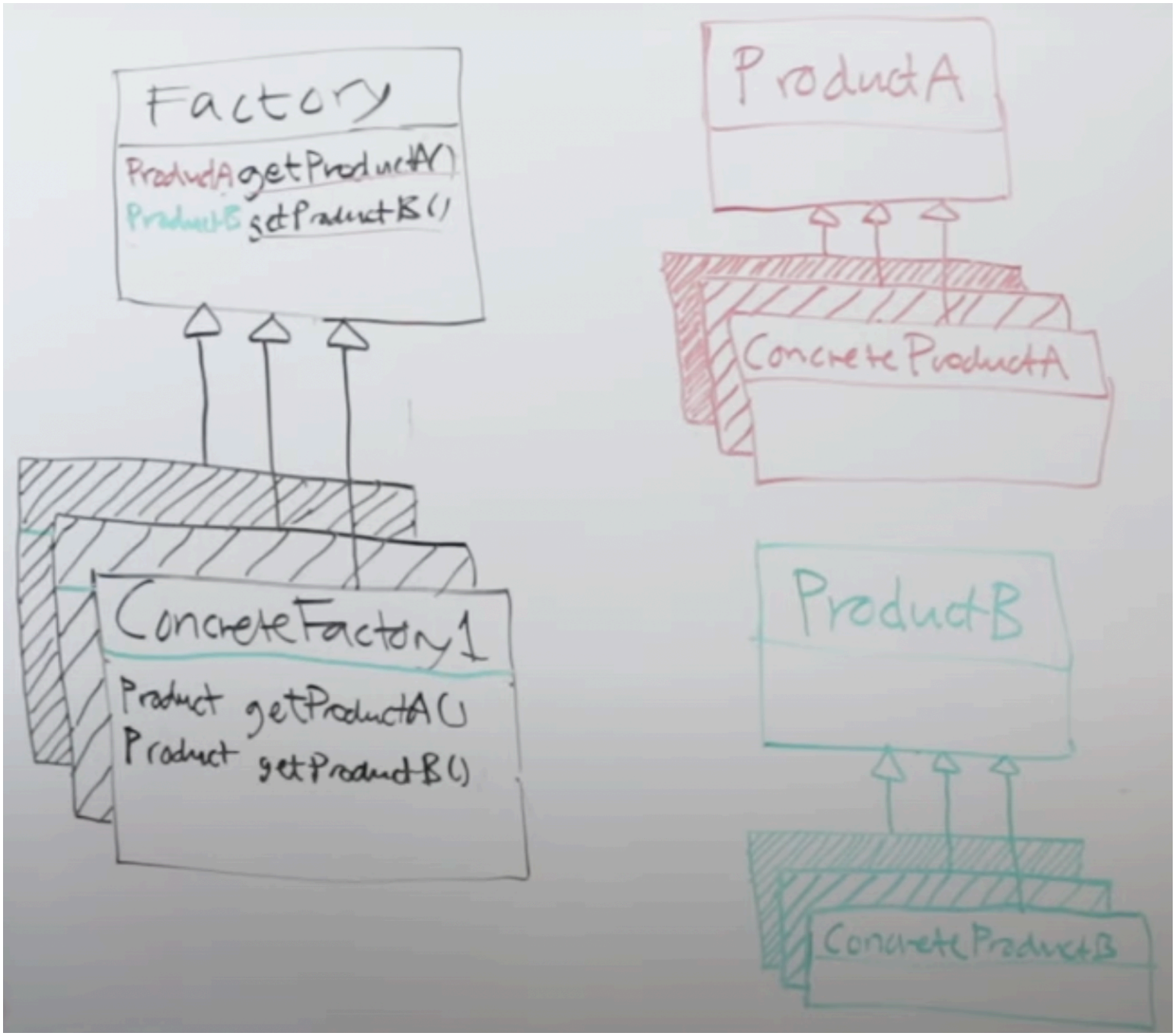
Let's say, you're building an application for Mac OS, and the same application has be deployed to Windows and to Linux as well. When we're talking of product, imagine that we're talking about some kind of UI component. So, there are some pairs of product that represent a single OS, and other pair of products another OS. So, within the context of some particular requirements, it might not necessarily make sense to mix certain objects with certain objects, So, in other words, it doesn't make sense to mix a Mac OS button with a Windows OS button.

So, you're creating buttons and texts using factory method, but you wouldn't necessarily have control of whether you accidentally mix something that's suitable for Mac, Windows or Linux. But with abstract factory pattern, you can have that control. It is because a factory creates two products. So, in our case, it was alert and a button. And there can be concrete factories of either a Mac OS factory or a Windows factory or a Linux factory. Which means that whenever we call getProductA(), we know that if we then later call getProductB(), we'll get a product B that is suitable in combination with product A, which means that they are of the same family and related products.

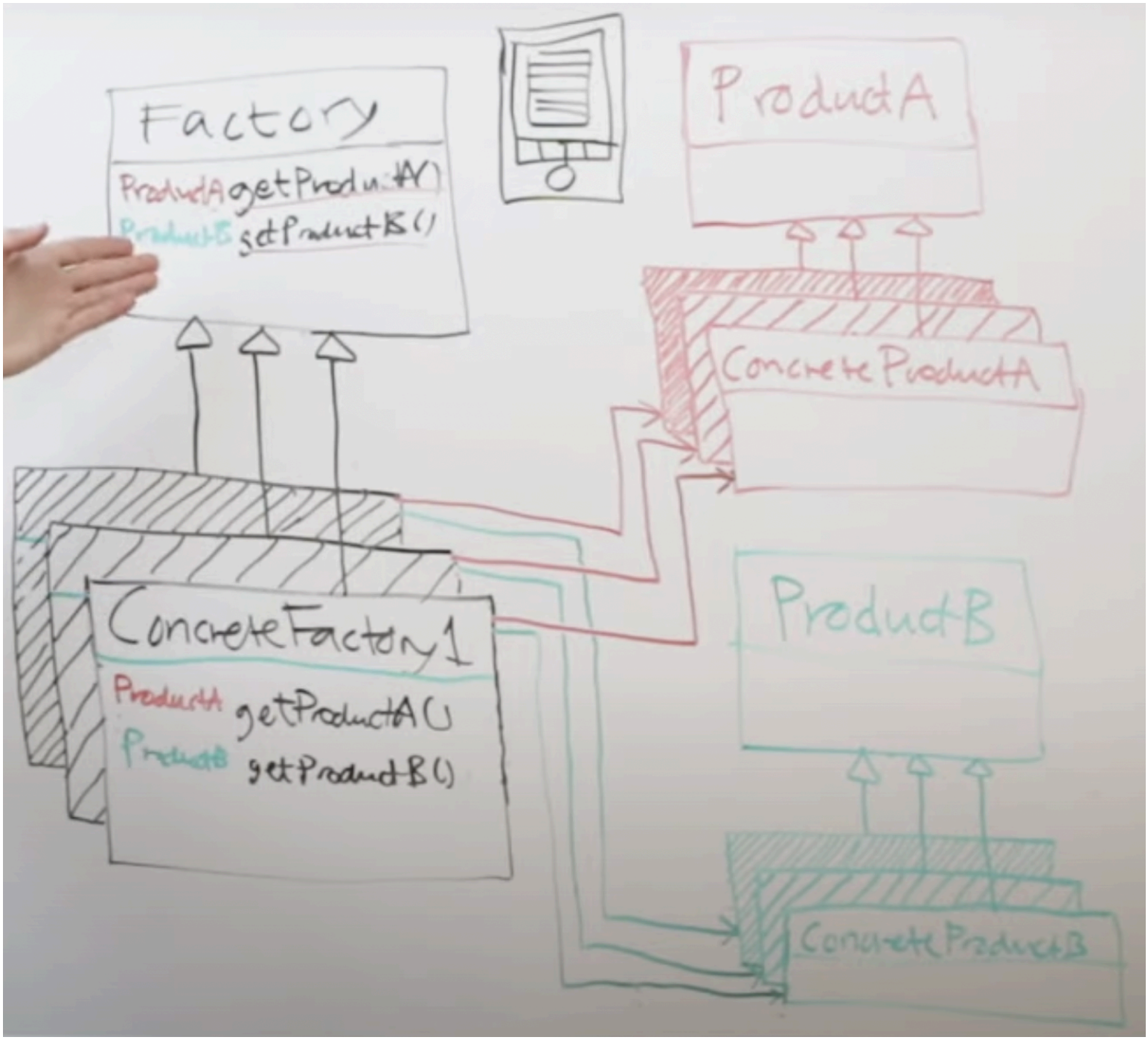
Because we have an abstract factory class, or a factory interface. So, we have mutiple concrete factories, we can imagine it as a bunch of factories behind each other.



Follow up from here can be that products that are in relation can have their own interfaces, and can be shown as different kind of products.



Think of abstract factory pattern as something that actually returns multiple products, and whether these products happen to be of the same type or happen to be of different types is a totally different concern.



Let's jump into the code example.

```
public interface Button {
String name();
}

public interface Label {
```

```
String name();
}

public interface Factory {
Button getButton();
Label getLabel();
}

public class BoldLabel implements Label {
private final String name;
BoldLabel(String os){
this.name = os + "BoldLabel";
}
public String name() {
return name;
}
}

public class ItalicLabel implements Label {
private final String name;
ItalicLabel(String os){
this.name = os + "ItalicLabel";
}
public String name() {
return name;
}
}
```

```
public class RoundedButton implements Button {
private final String name;
RoundedButton(String os){
name = os + "RoundedButton";
}
public String name() {
return name;
}
}

public class ShadowButton implements Button {
private final String name;
ShadowButton(String os){
name = os + "ShadowButton";
}
public String name() {
return name;
}
}

import java.util.Random;
public class MacOSFactory implements Factory{
public Button getButton() {
return switch(new Random().nextInt(2)){
case 0 -> new RoundedButton("MacOS");
case 1 -> new ShadowButton("MacOS");
default -> null;
};
```

```
}
public Label getLabel() {
return switch(new Random().nextInt(2)){
case 0 -> new BoldLabel("MacOS");
case 1 -> new ItalicLabel("MacOS");
default -> null;
};
}
}

import java.util.Random;
public class WindowsOSFactory implements Factory{
public Button getButton() {
return switch(new Random().nextInt(2)){
case 0 -> new RoundedButton("WindowsOS");
case 1 -> new ShadowButton("WindowsOS");
default -> null;
};
}
public Label getLabel() {
return switch(new Random().nextInt(2)){
case 0 -> new BoldLabel("WindowsOS");
case 1 -> new ItalicLabel("WindowsOS");
default -> null;
};
}
}
```

```
public class Main {
public static void main(String[] args) {
Factory factory;
factory = new MacOSFactory();
System.out.println(factory.getLabel().name());
System.out.println(factory.getButton().name());
factory = new WindowsOSFactory();
System.out.println(factory.getLabel().name());
System.out.println(factory.getButton().name());
}
}

MacOSItalicLabel
MacOSShadowButton
WindowsOSBoldLabel
WindowsOSShadowButton

Process finished with exit code 0
```

This page have been intentionally left blank

NULL OBJECT PATTERN

The keyword null or the value null is what we use to represent nothingness. So, the notion of null is the notion of not having a value.

The inventor of the idea of null is Tony Hoare, and he has apparently apologized for inventing the null and has called it his billion dollor mistake.

Essentially, the point is that null is tricky, nothingness is tricky, because whenever we have something which is knowable, meaning it either be the value or it can be null, which is not the same as exactly knowing it. This is why some langauges separate the notion of an int from unknowable int. So, if we say unknowable int, then it definitely has to have a number, we can't have an null int because if its knowable, then it's either null or an int.

Null is powerful because sometimes we just want to specify that something is nothing, and sometimes you just want to have a special value that you want to be able to check for so that you can terminate a loop for example in traversing a linked list. So, whenever we introduce null, we either have a thing or we don't which means that there are necessarily two branches through the program.

So, if something is null that means there is one path through the program where we do something with it if it's not null and there is another path where we do something else if it is null.

But here, as soon as something is knowable, we have to check whether it's null or not. Like as we have parameters in a function, and if someone passes null values into it, then we have to check for each value in the function before we use it. And just like this, every method is starting up with checking whether the values it's going to evaluate are null or not.

But enter the null object pattern. Why do we need the null object pattern?

One of the major reasons we use objects because it gives us polymorphism. It allows us to say that, we have this thing which is of some general type but I'm not sure what specific concrete type it is of, but if we treat it as of some kind of type, then we can dispatch method calls, which will necessarily respond to those methods, because it is of some parent type. And there is a likelihood that the parent type may be null, and to handle it, we can use polymorphism with the null type.

In the similar way, if we have different classes that implements the same interface, and those different classes represents different objects within the same kind, which means within the same interface, then there is also a possibility that we may

create an object which represents null type, which means nothing, which means no object.

Like in some scenarios, it's completely logical to have a no object, for example, if we remember the duck example in our strategy pattern, we used to say some ducks have no flying behaviour and no quack behaviour, there we can use the null object pattern to represent these behaviours which are actually nothing and represents no behaviour.

So, it's completely sensible in some scenarios to have a knowing state, a state that actually does nothing because we want to avoid to do this null checking, so we want to be able to inject something that behaves just as all of the other things behave but that simply does nothing or does sort of the default thing.

For example consider a stickman game with buttons up down left right to move the stick. Now, if the player presses those buttons the player is in the moving state with some direction, but if those buttons are released, it goes into the no moving state, which freezes the stickman right where it is, which means that this state is the absense of the moving state, where possibly the value of movement could be null, and for that we could use the null object patten to represent no moving state as an object that does nothing, or in this example, does not lets the stick move.

This page have been intentionally left blank

ITERATOR PATTERN

The iterator pattern is all about iterating over the items in a collection. Inumerating all of the items in a collection. What is a collection? A collection of set of things that you define yourself.

Whenever you have a collection, we have to iterate through them sequentially. But the thing is how to iterate over somethig? How to enumarate something? How to traverse a structure?

If we have some kind of collection, and it contains some items, and I want to be able to iterate over it from the outside, and go through each of the items.

So, whenever I want to iterate through them, I shouldn't have to care about the structure of this underlying collection, either we build it, or somebody else builds it. I should be able to simply say, give me each of the items one by one.

The idea is that if I can iterate over the contents of this collection with an interface that other collections also use then I can suddenly treat different collections uniformly or transparently.

In iterator, we don't have to care about which particular iterateable thing we have, for example, whether it's a List or an ArrayList, they are iterated in the same way.

The iterator pattern is already implemented in the java collections framework, but we can have some complex data structure that we can create on our own, and can use iterator pattern on it, to traverse it uniformly.

The benefit of iterator pattern is that you don't have to expose your complex structure for someone else to just iterate over it. Inevitably, there is some computation that it has to go into to actually flatten the complex structure and iterate over it.

Another benefit is that with iterator pattern, we're not giving the whole collection to the client at once, what we're giving is one item, then next item and so on. So, that has an encapsulation benefit and information hiding benefit. Collections does not have to expose their underlying representation.

Another benefit is lazy evaluation, we can stop the evaluation at anytime, the client has full control over how it wants to iterate over the collection. Therefore, we can actually contruct infinite collections with the control of iteration, so that the client can keep track of where collection is in the iteration, can pause at that location, and can also continue later on.

Whenever we're using the iterator pattern, we're going to have a pointer that points to whichever item we're at now, so when the client asks for the next item, the pointer returns the item where it is, and goes to the next items on its own.

In some languages, we have features like innumerable and the enumerator. For example, if house class is innumerable and returns and has a method that returns an enumeratee, that means we can use for each construct even though it's not a collection, and it's essentially because of the iterator pattern.

```
House h = new House();
for each(Piece p in h){}
```

Or we could do something like this with the Iterator. Here, we're manuallt controlling the iterator pointer because it gives us the possibility of getting the same value multiple times if we ever needed.

```
Iterator i = h.getItems();
while(i.hasNext()){
i.next();
i.currentItems();
}
```

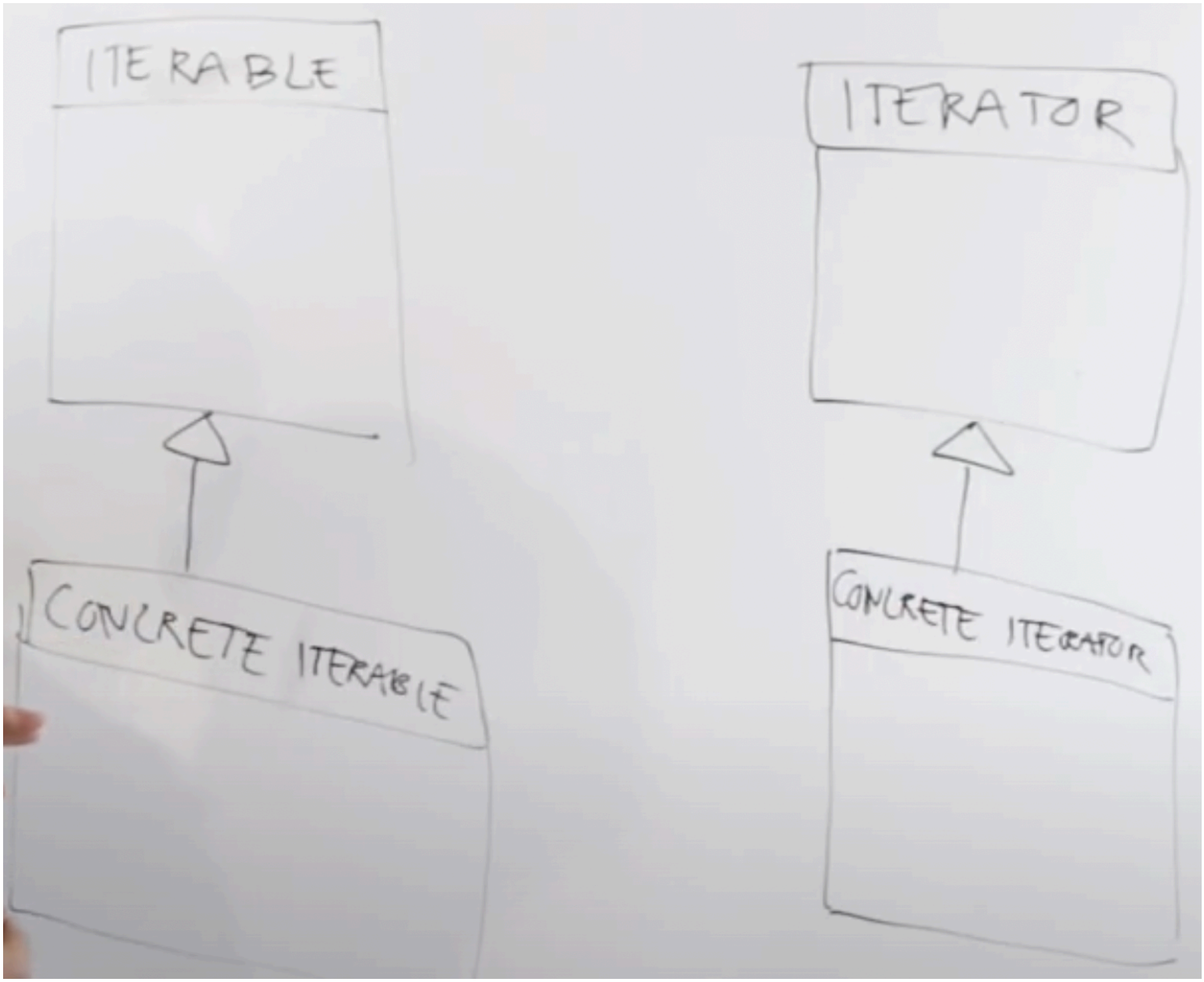
Now, let's look at the definition of the Iterator pattern.

" The iterator pattern provides a way to access the elements of an aggregate object sequentially without exposing its undering representation. "

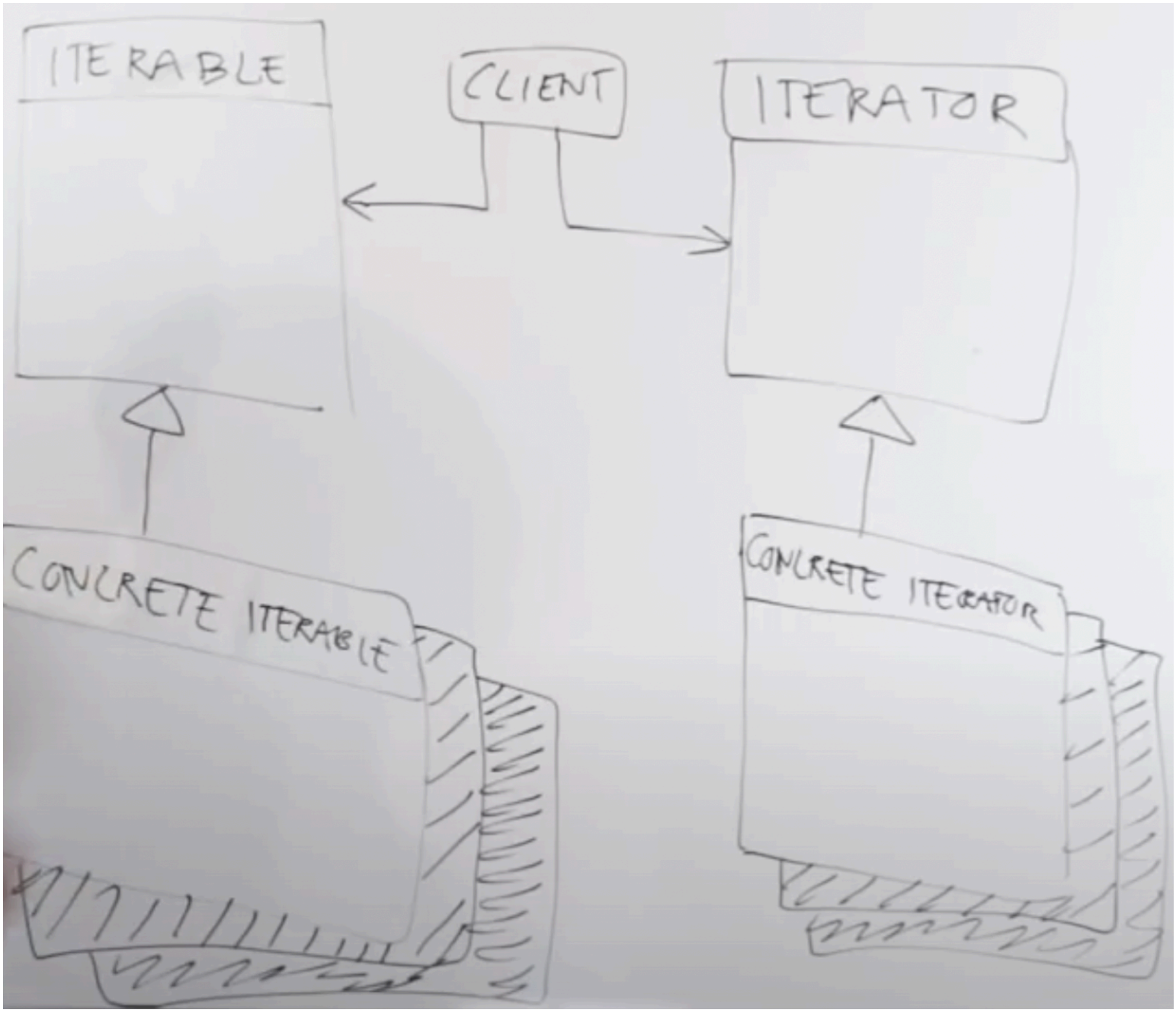
In other words, an object that aggregates a number of things, which means an object that contains a number of things, and our end goal is that we want to access them sequentially one after another, but we want to do so without exposing the undering representation, which means without having to care about how this collection is actually structured.

Let's look at the UML diagram now,

First, we have an Iterable, also known as an enumerable, or an aggregate, it's the thing that aggregates other things, it's the thing that contains things, it's essentially the collection. And then we have the concrete Iterable, which is an iterable object that implements the iterable interface, that's one inheritance hierarchy. The second inheritance hierarchy is that we have an Iterator also known as enumerate which is implemented by or inherited from or rather there's a thing that inherits from the iterator and that thing we simply call a concrete iterator.



We can actually have multiple concrete iterables that implements a common iterable interface, and we can also have multiple concrete iterators that implements a common iterator interface. We also have the client that has a iterable, and has a iterator as well.



So, what methods does the iterable require? If it is an Iterable, then it should have a method getIterator() that returns an iterator.

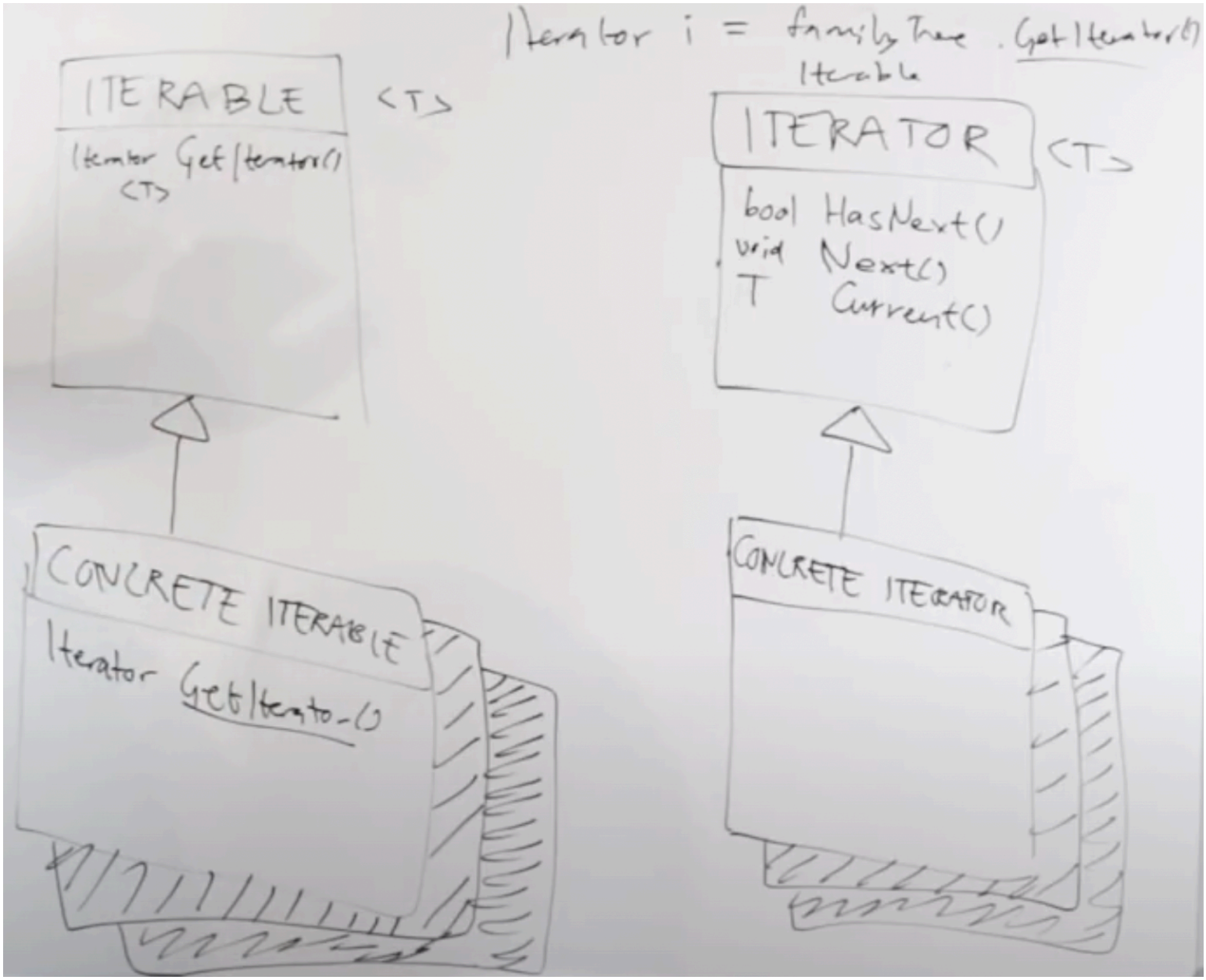
Iterator i = h.getIterator();

Now, an iterator generally has three methods. First is a method that returns bool which is called hasNext(). In other words, we need to be able to know whether this iterator has iterated over

the iterable all the way to the end whatever it has iterated. So, it checks whether we've seen all of the items already or still there are left? Another method is next() which actually takes a void type but mutates the state of the iterator.

Whenever you have a method that mutates something, you could also redesign it as an immutable version of that method, so instead of having a void method called next(), we could have a method that returns an iterator so that the iterator could return a new instance of itself with the pointer moved the the next loaction, that would be a way of avoiding mutation within this iterator.

And then we have a method that finally returns the item where the pointer is currently, which is called current(), what it returns completely depends on our scenario, so we call have a placeholder like Object type., or we can even use generics to define the type the current() of iterator will return.


Iterator i = familyTree.GetIterator()
Iterable
ITERABLE <T>
Iterator GetIterator()
<T>
ITERATOR <T>
bool HasNext()
void Next()
T Current()
CONCRETE ITERABLE
Iterator GetIterator()
CONCRETE ITERATOR