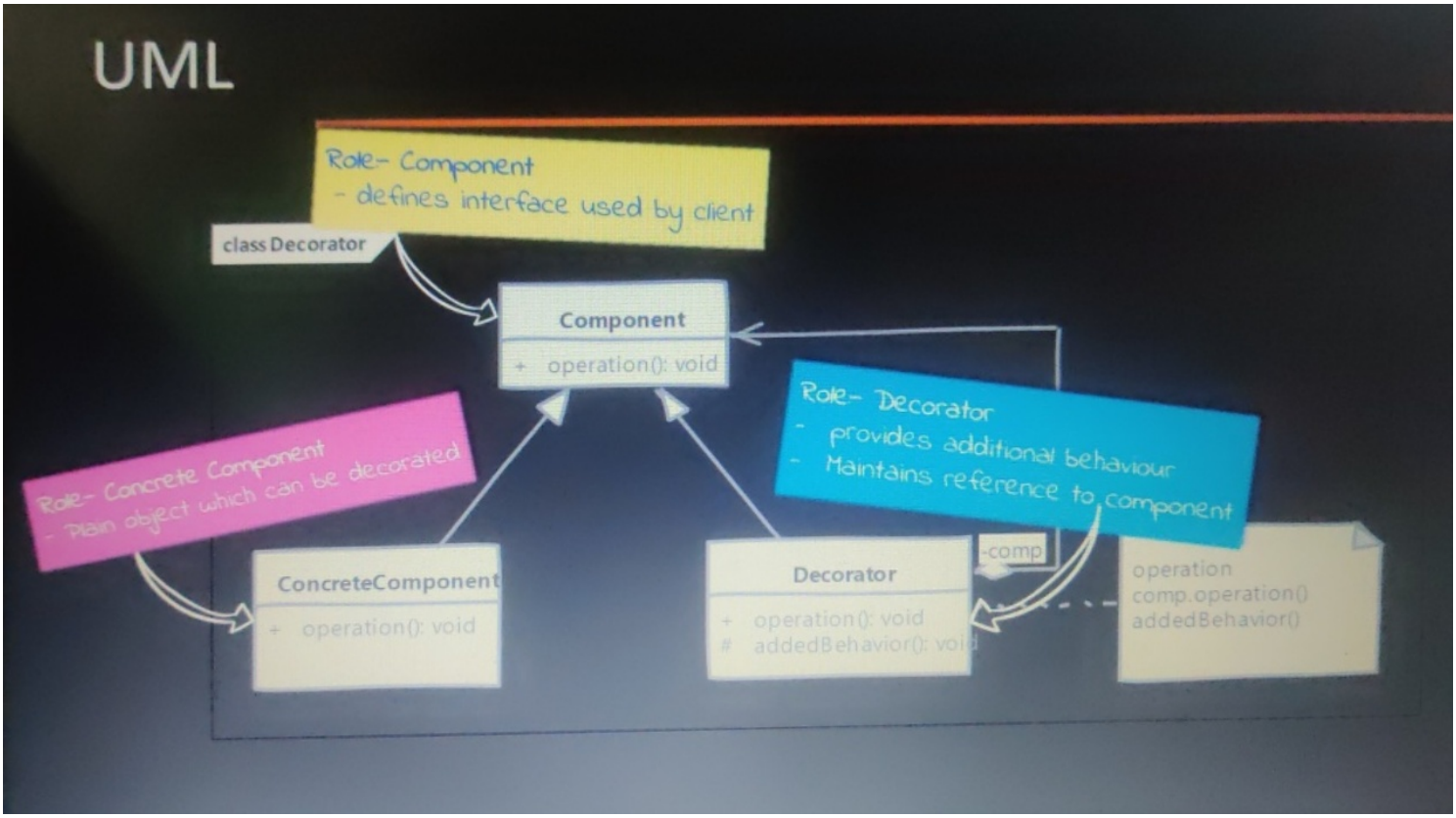

Decorator
Structural Design Patterns
Design Patterns in Java

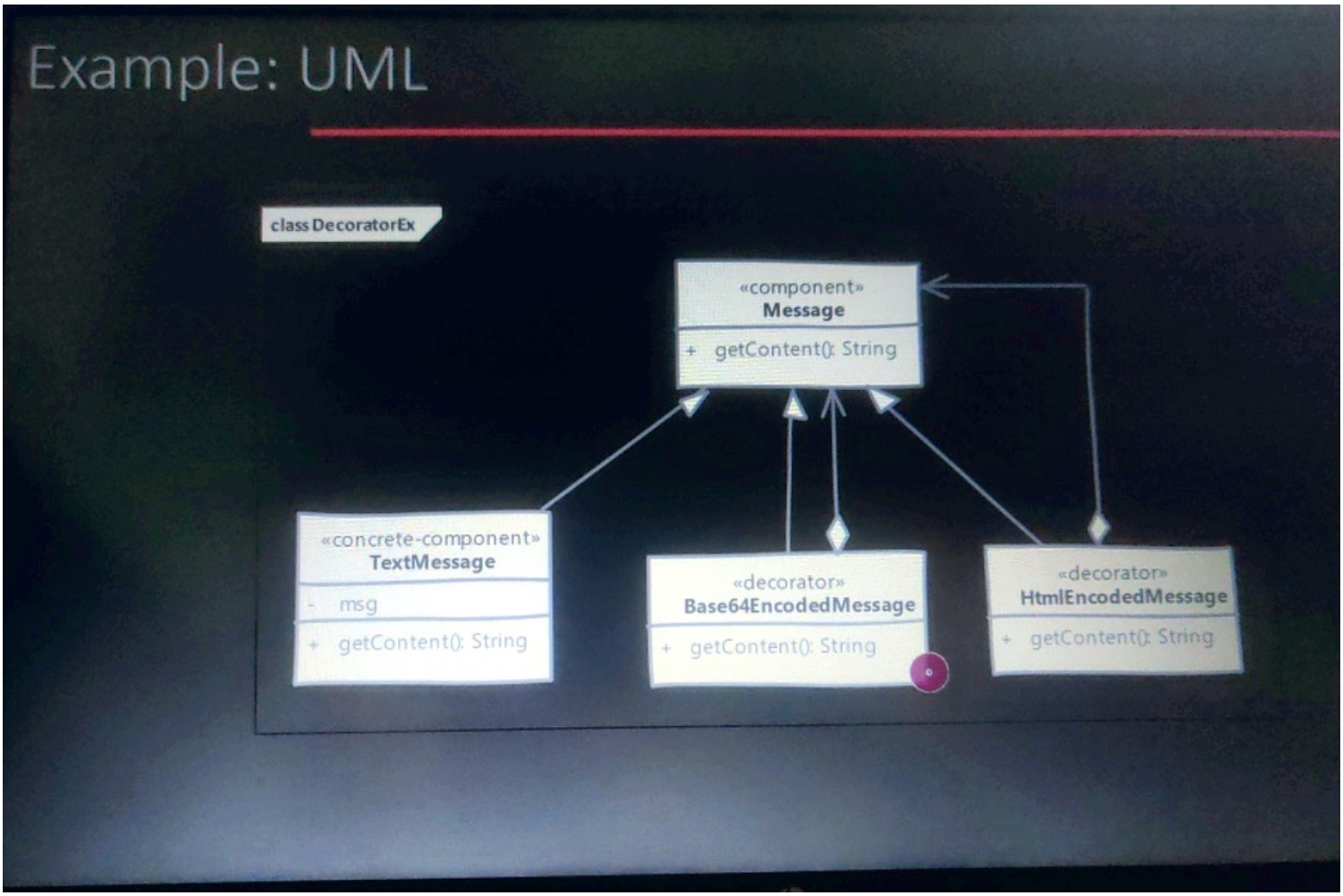
# What is Decorator?

- When we want to enhance behaviour of our existing object dynamically as and when required then we can use decorator design pattern.
- Decorator wraps an object within itself and provides same interface as the wrapped object. So the client of original object doesn't need to change.
- A decorator provides alternative to subclassing for extending functionality of existing classes.


UML
Role- Component
- defines interface used by client
class Decorator
Component
+ operation(): void
Role- Decorator
- provides additional behaviour
- Maintains reference to component
Role- Concrete Component
- Plain object which can be decorated
ConcreteComponent
+ operation(): void
Decorator
+ operation(): void
# addedBehavior(): void
-comp
operation
comp.operation()
addedBehavior()

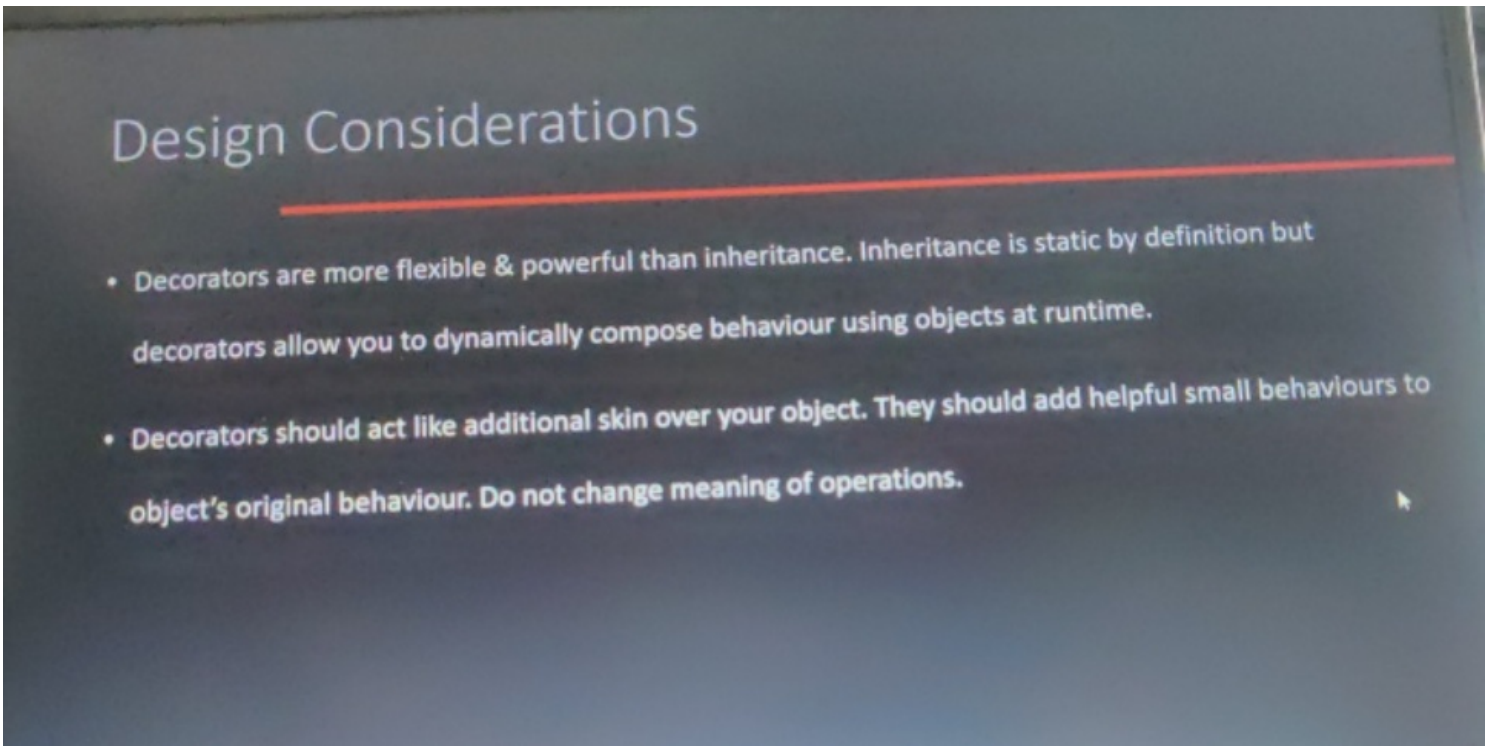
# Implement a Decorator

- We start with our component.
  - Component defines interface needed or already used by client.
  - Concrete component implements the component.
  - We define our decorator. Decorator implements component & also needs reference to concrete component.
  - In decorator methods we provide additional behaviour on top that provided by concrete component instance
- Decorator can be abstract as well & depend on subclasses to provided functionality.


Example: UML
class DecoratorEx
«component»
Message
+ getContent(): String
«concrete-component»
TextMessage
- msg
+ getContent(): String
«decorator»
Base64EncodedMessage
+ getContent(): String
«decorator»
HtmlEncodedMessage
+ getContent(): String

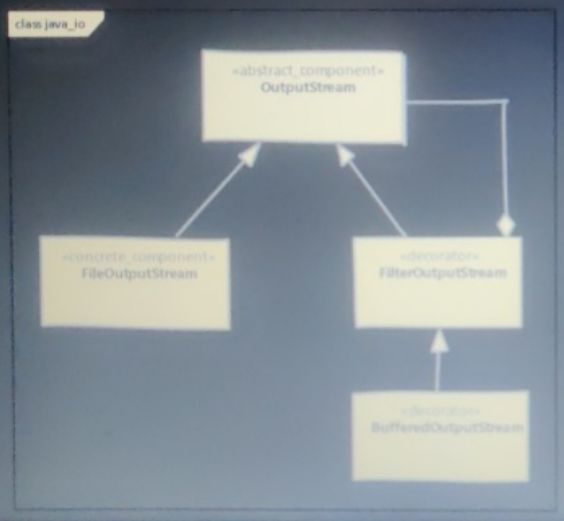
# Implementation Considerations

- Since we have decorators and concrete classes extending from common component, avoid large state in this base class as decorators may not need all that state.
- Pay attention to equals and hashCode methods of decorator. When using decorators, you have to decide if decorated object is equal to same instance without decorator.
- Decorators support recursive composition, and so this pattern lends itself to creation of lots of small objects that add "just a little bit" functionality. Code using these objects becomes difficult to debug.

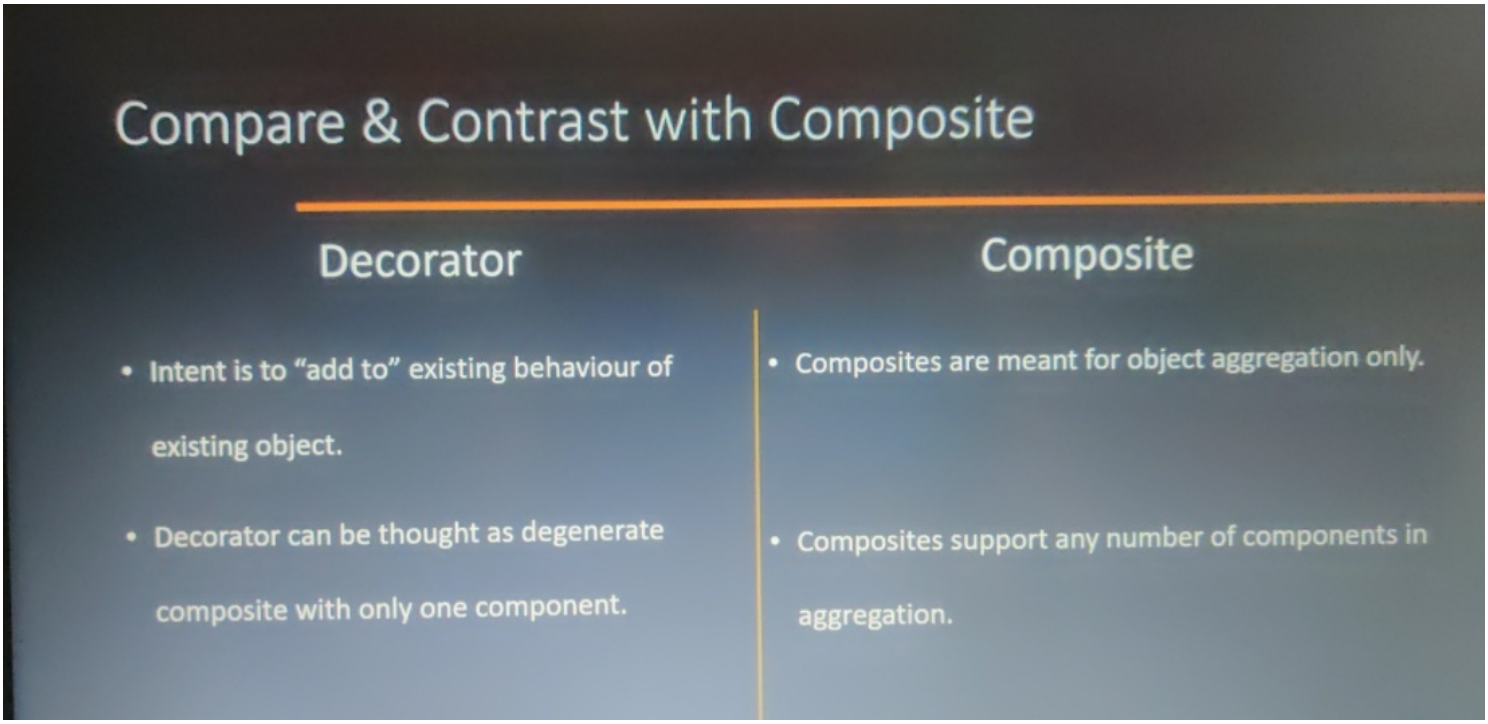

Design Considerations
• Decorators are more flexible & powerful than inheritance. Inheritance is static by definition but decorators allow you to dynamically compose behaviour using objects at runtime.
• Decorators should act like additional skin over your object. They should add helpful small behaviours to object's original behaviour. Do not change meaning of operations.

# Example of a Decorator

- Classes in Java's I/O package are great examples of decorator pattern.
- For example the java.io.BufferedOutputStream class decorates any java.io.OutputStream object and adds buffering to file writing operation. This improves the disk i/o performance by reducing number of writes.

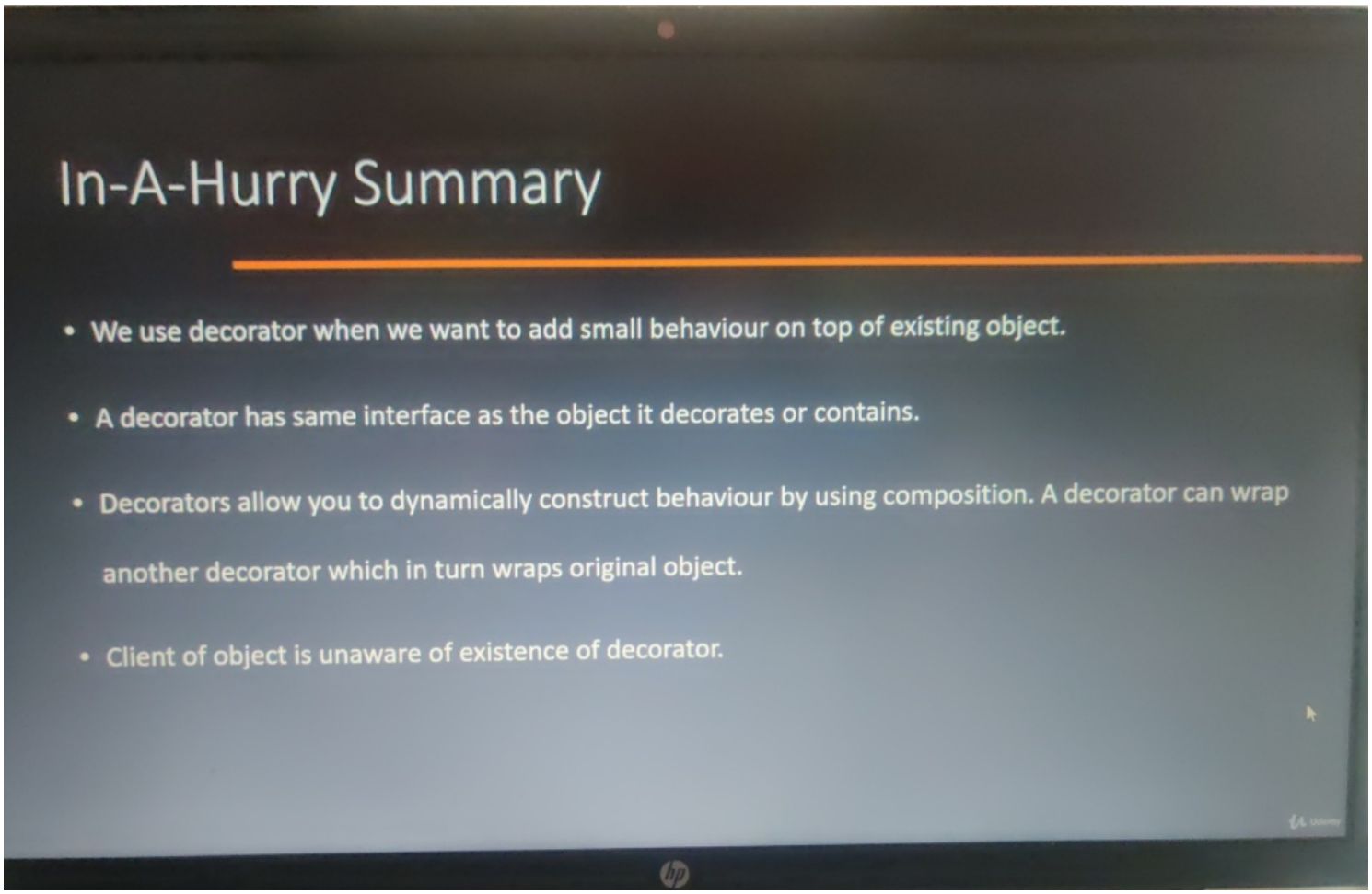


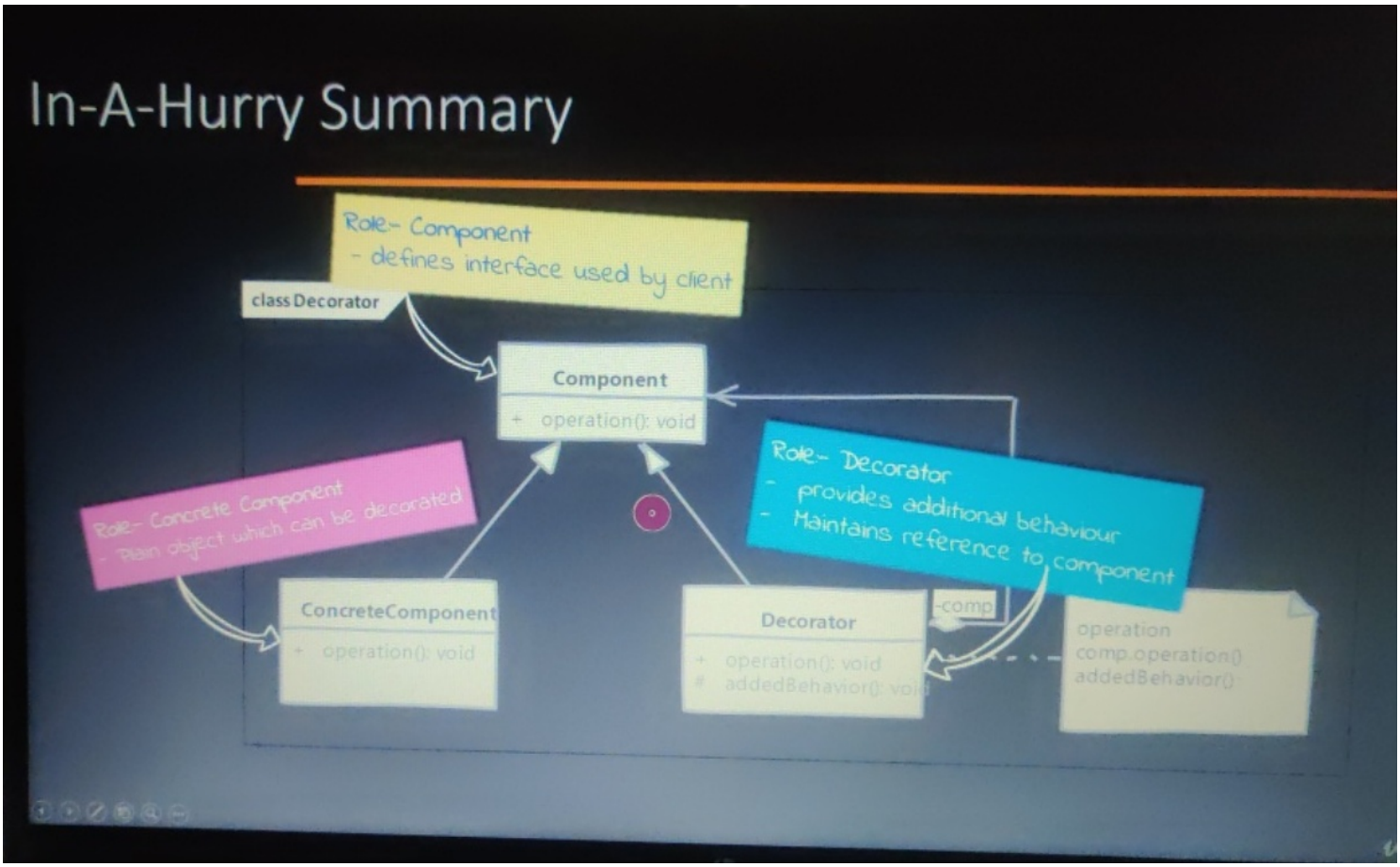

```
try (OutputStream os = new BufferedOutputStream(
        new FileOutputStream("xfiles_mulder_notes.txt")
        )){
    os.write('x');
    os.flush();
}
```


Compare & Contrast with Composite
Decorator
Composite
• Intent is to "add to" existing behaviour of existing object.
• Decorator can be thought as degenerate composite with only one component.
• Composites are meant for object aggregation only.
• Composites support any number of components in aggregation.

# Pitfalls

- Often results in large number of classes being added to system, where each class adds a small amount of functionality. You often end up with lots of objects, one nested inside another and so on.
- Sometimes new comers will start using it as a replacement of inheritance in every scenario. Think of decorators as a thin skin over existing object.


In-A-Hurry Summary
• We use decorator when we want to add small behaviour on top of existing object.
• A decorator has same interface as the object it decorates or contains.
• Decorators allow you to dynamically construct behaviour by using composition. A decorator can wrap another decorator which in turn wraps original object.
• Client of object is unaware of existence of decorator.


In-A-Hurry Summary
Role- Component
- defines interface used by client
class Decorator
Component
+ operation(): void
Role- Concrete Component
- Plain object which can be decorated
Role- Decorator
- provides additional behaviour
- Maintains reference to component
ConcreteComponent
+ operation(): void
Decorator
+ operation(): void
# addedBehavior(): void
-comp
operation
comp.operation()
addedBehavior()