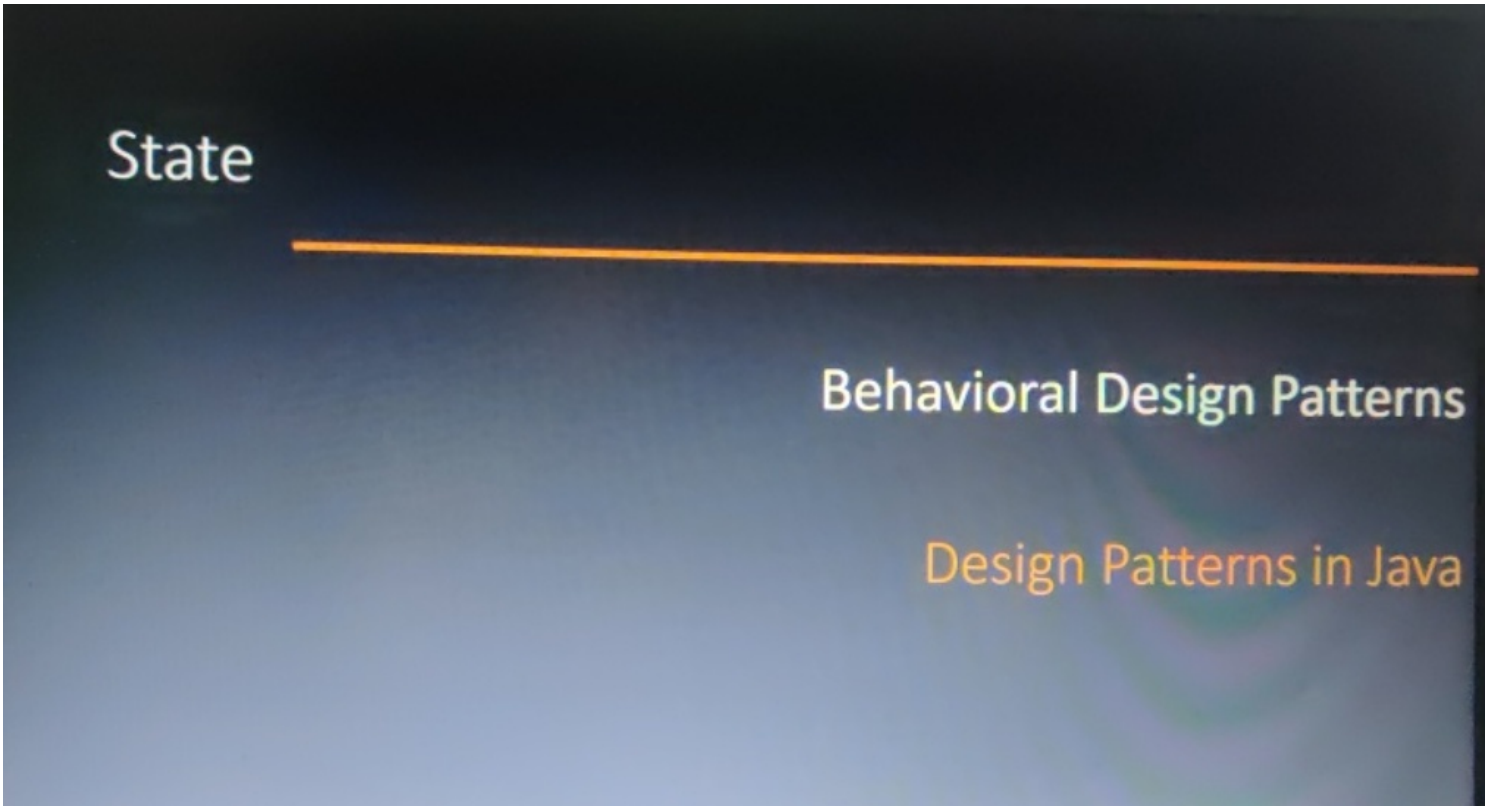
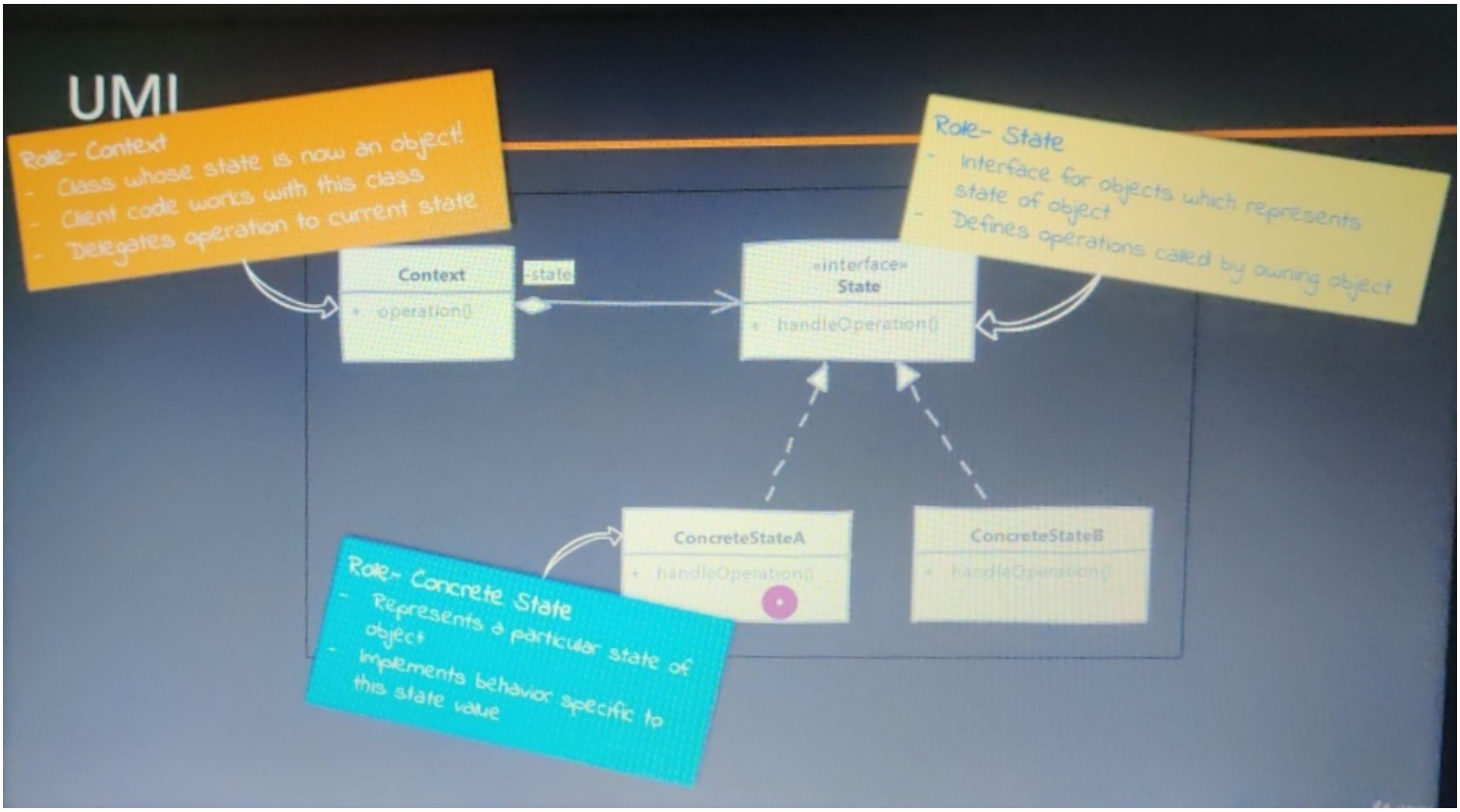

State
Behavioral Design Patterns
Design Patterns in Java

What is a State design pattern?

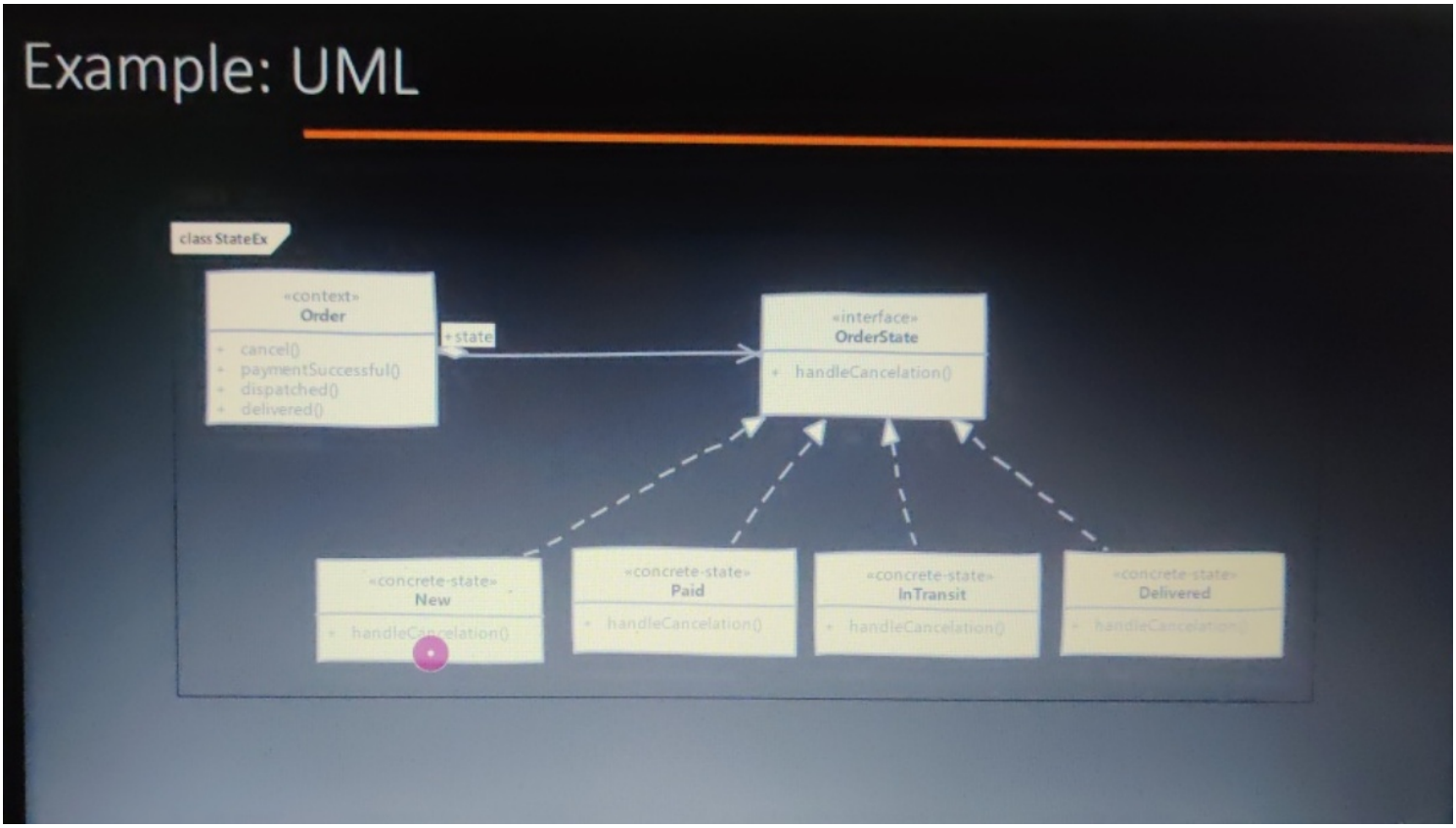
- State design pattern allows our objects to behave differently based on its current internal state.
- This pattern allows to define the state specific behaviors in separate classes.
- Operations defined in the class delegate to the current state object's implementation of that behavior.
- State transitions can be triggered by states themselves in which case each state knows about at least one other state's existence.
- A benefit of this pattern is that new states and thus new behaviors can be added without changing our main class.


UML
Role- Context
- Class whose state is now an object!
- Client code works with this class
- Delegates operation to current state
Role- State
- Interface for objects which represents state of object
- Defines operations called by owning object
Context
+ operation()
-state
«interface»
State
+ handleOperation()
ConcreteStateA
+ handleOperation()
ConcreteStateB
+ handleOperation()
Role- Concrete State
- Represents a particular state of object
- Implements behavior specific to this state value

Implement State pattern

- Identify distinct values for state of our object (context). Each state value will be a separate class in our implementation. These classes will provide behavior specific to the state value they represent.
- In our main/context class method implementations we'll delegate the operation to current state object.
- We have to decide how our state transition is going to happen. States can themselves transition to next state based on input received in a method. Other option is context itself can initiate transition.
- Client interacts with our main class or context and is unaware of existence of state.

Example: UML


class StateEx
«context»
Order
+ cancel()
+ paymentSuccessful()
+ dispatched()
+ delivered()
+state
«interface»
OrderState
+ handleCancelation()
«concrete-state»
New
+ handleCancelation()
«concrete-state»
Paid
+ handleCancelation()
«concrete-state»
InTransit
+ handleCancelation()
«concrete-state»
Delivered
+ handleCancelation()

Implementation Considerations

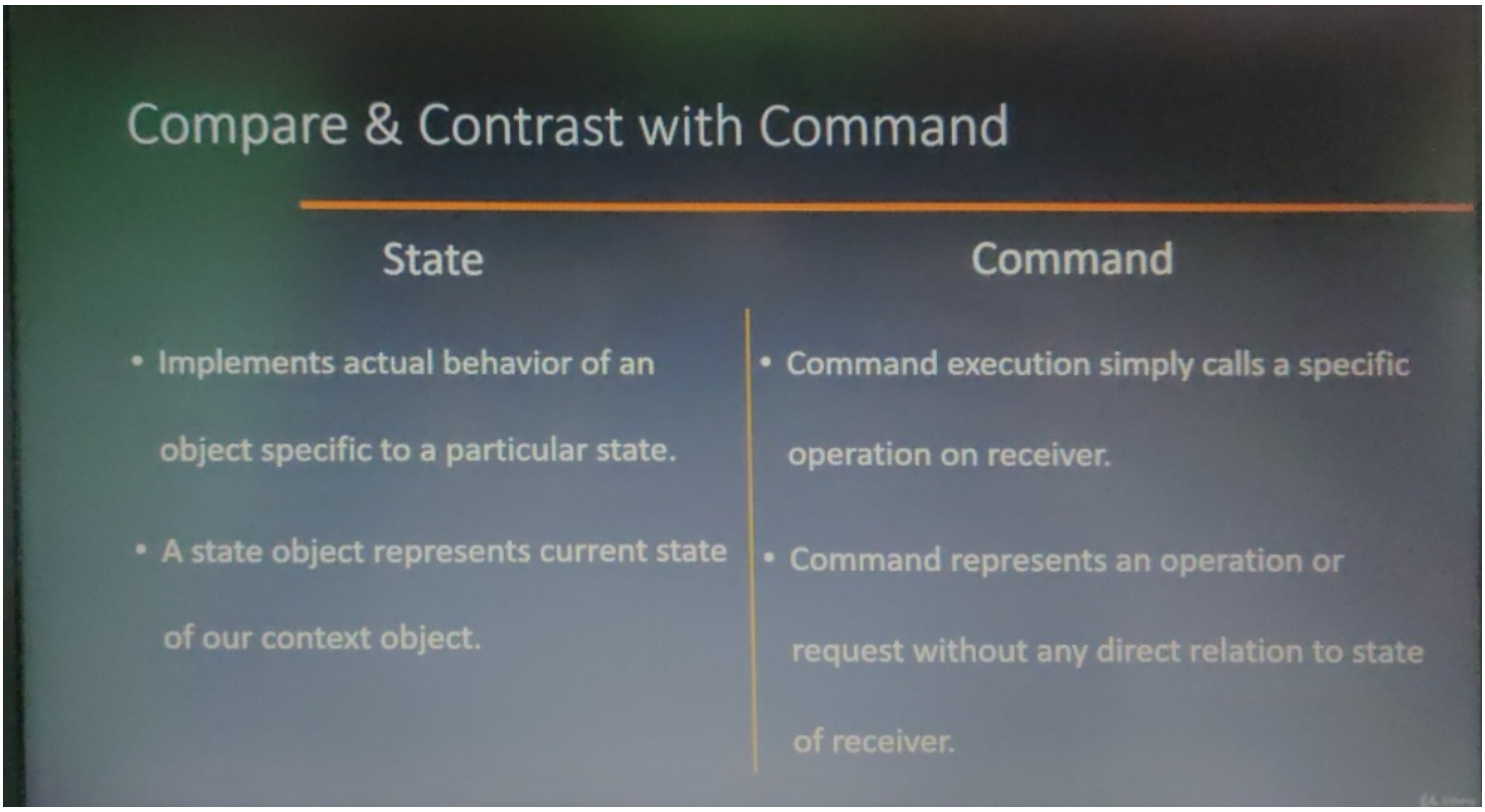
- In some implementations clients themselves can configure context with initial state. However after that the state transition is handled either by states or context.
- If state transitions are done by state object itself then it has to know about at least one state. This adds to the amount of code change needed when adding new states.

Design Considerations

- Using flyweight pattern we can share states which do not have any instance variables and only encapsulate behaviour specific to that state.
- State design pattern is not same as a state machine. A state machine in loose terms focuses on state transitions based on input values & using some table to map these inputs to states. A state design pattern focuses on providing a behaviour specific to a state value of context object.

Examples of State pattern

- One of the examples of state pattern can be found in JSF(Java Server Faces) framework's LifeCycle implementation.
- FacesServlet will invoke execute & render methods of LifeCycle. LifeCycle instance in turn collaborates with multiple "phases" to execute a JSF request. Here each phase represents a state in state pattern.
- JSF has six phases: RestoreViewPhase, ApplyRequestValues, ProcessValidationsPhase, UpdateModelValuesPhase, InvokeApplicationPhase, and RenderResponsePhase
- The LifeCycle also provides an object of FacesContext to these phases to help in providing state specific behavior.
- In case you are not familiar with JSF, then you can always fall back on the order processing example that we studied.

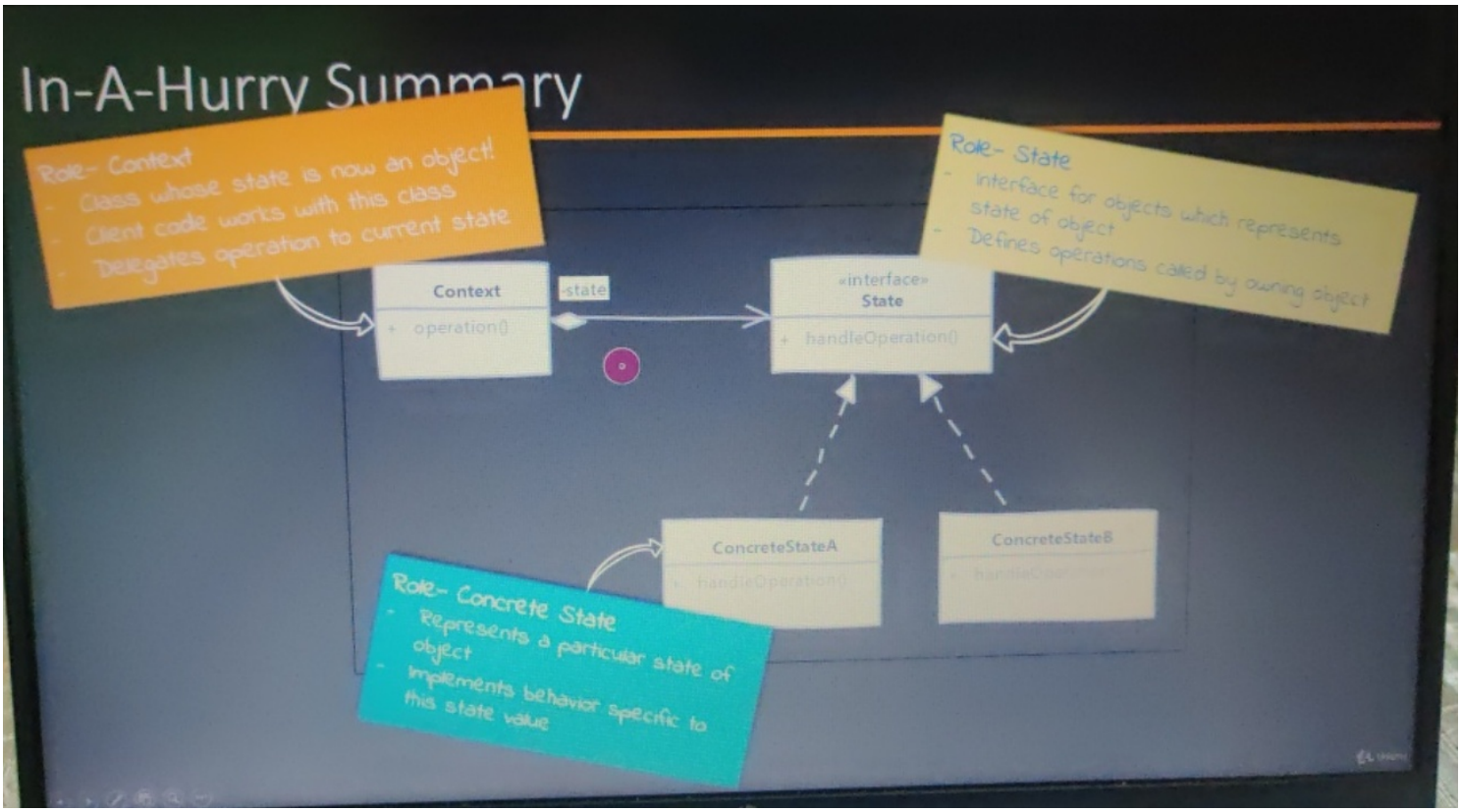

Compare & Contrast with Command
State
Implements actual behavior of an object specific to a particular state.
A state object represents current state of our context object.
Command
Command execution simply calls a specific operation on receiver.
Command represents an operation or request without any direct relation to state of receiver.

Pitfalls

- A lot more classes are created for providing functionality of context & all those need unit testing as well.
- State transitions can be a bit tricky to implement. This becomes more complicated if there multiple possible states to which object can transition from current state. And if states are responsible for triggering transitions then we have a lot more coupling between states.
- We may not realize all possible states we need at the beginning of our design. As our design evolves we may need to add more states to handle a particular behavior.

In-A-Hurry Summary

- If we have an object whose behavior is completely tied to its internal state which can be expressed as an object we can use the state pattern.
- Each possible state ***value*** now becomes a class providing behavior specific to a state value.
- Our main object (aka context) delegates the actual operation to its current state. States will implement behavior which is specific to a particular state value.
- Context object's state change is explicit now, since we change the entire state object.
- State transitions are handled either by states themselves or context can trigger them.
- We can reuse state objects if they don't have any instance variables and only provide behavior.


In-A-Hurry Summary
Role- Context
- Class whose state is now an object!
- Client code works with this class
- Delegates operation to current state
Role- State
- Interface for objects which represents state of object
- Defines operations called by owning object
Context
+ operation()
state
«interface»
State
+ handleOperation()
ConcreteStateA
ConcreteStateB
Role- Concrete State
- Represents a particular state of object
- Implements behavior specific to this state value