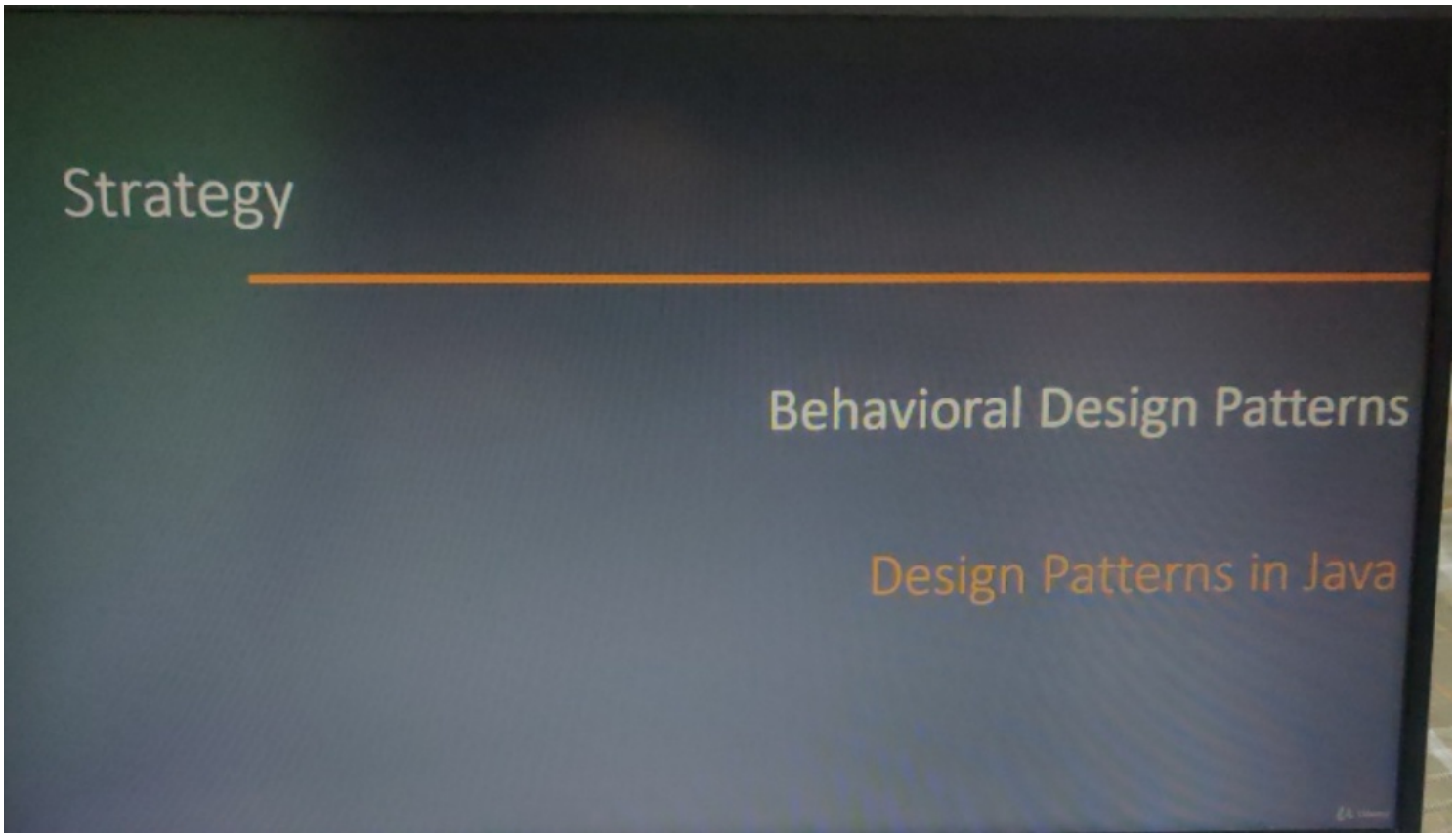
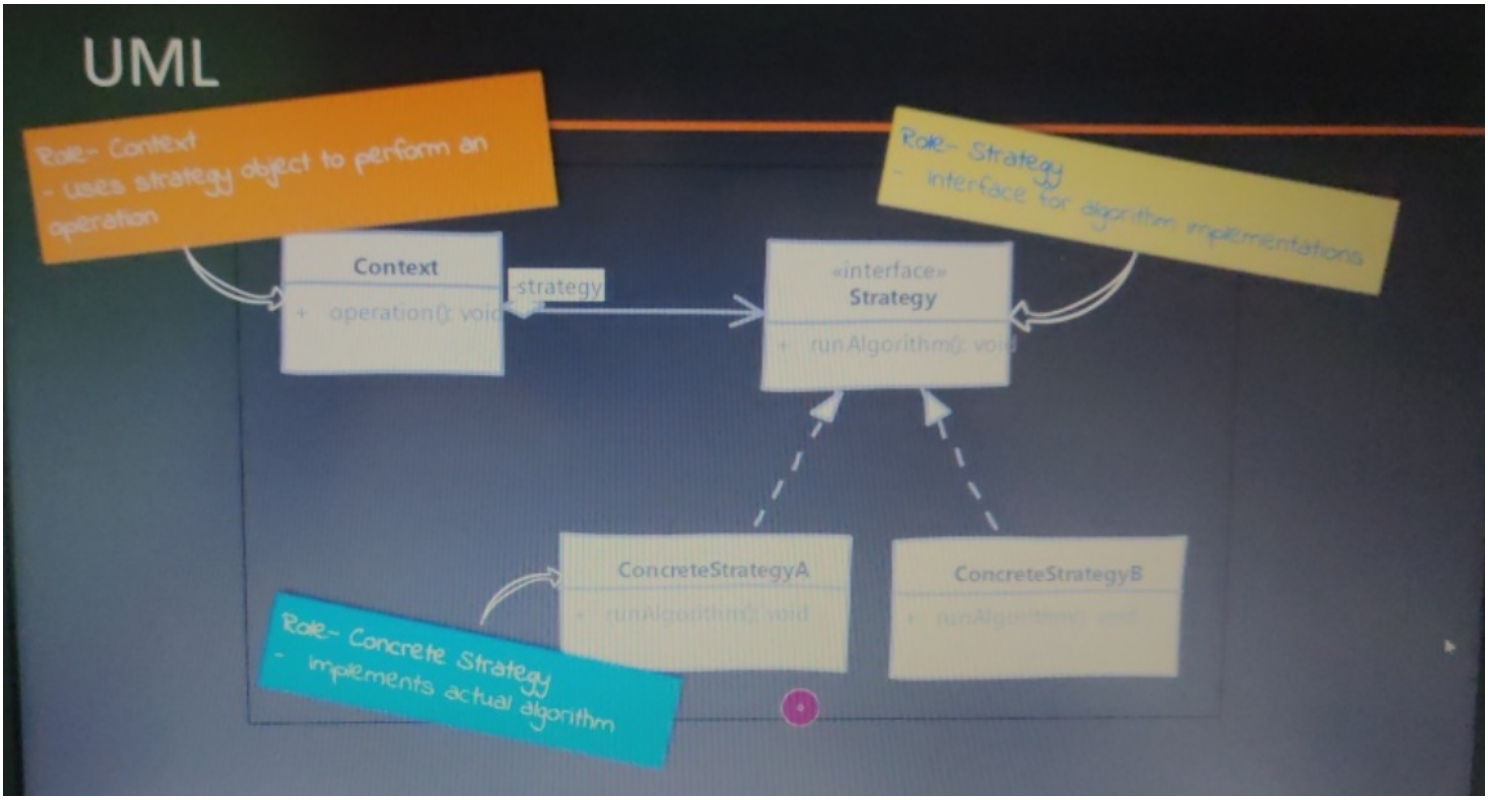

Strategy
Behavioral Design Patterns
Design Patterns in Java

What is a Strategy design pattern?

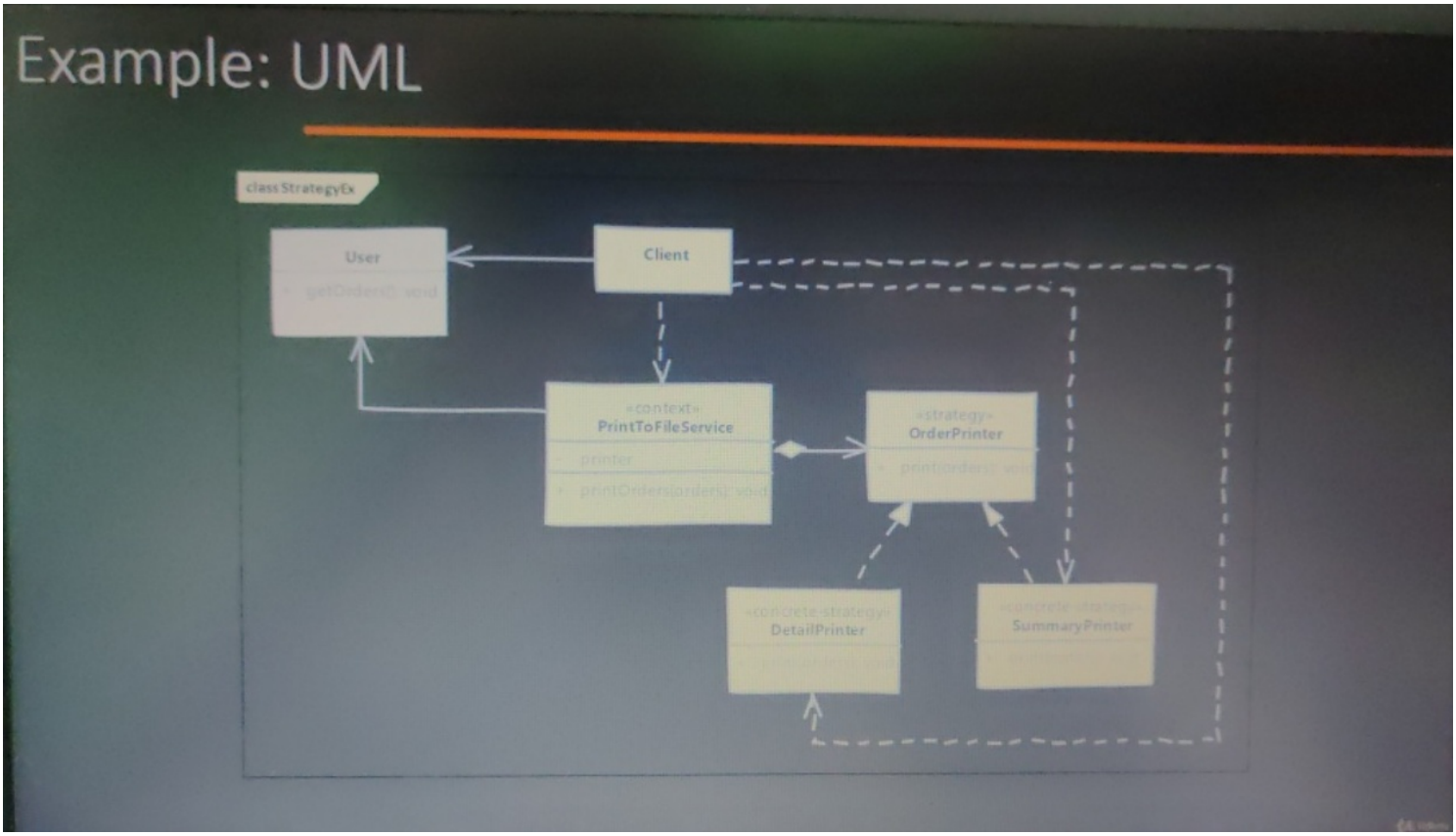
- Strategy pattern allows us to encapsulate an algorithm in a class. So now we can configure our context or main object with an object of this class, to change the algorithm used to perform given operation.
- This is really helpful if you have many possible variations of an algorithm.
- A good indication for applicability of strategy pattern is if we find different algorithms/behaviors in our methods which are selected with conditional statements like if-else or switch-case.
- Strategy classes are usually implemented in an inheritance hierarchy so that we can choose any one implementation and it'll work with our main object/context as the interface is same for all implementations.


UML
Role- Context
- uses strategy object to perform an operation
Role- Strategy
- interface for algorithm implementations
Context
+ operation(): void
-strategy
«interface»
Strategy
+ runAlgorithm(): void
ConcreteStrategyA
ConcreteStrategyB
Role- Concrete Strategy
- implements actual algorithm

Implement Strategy pattern

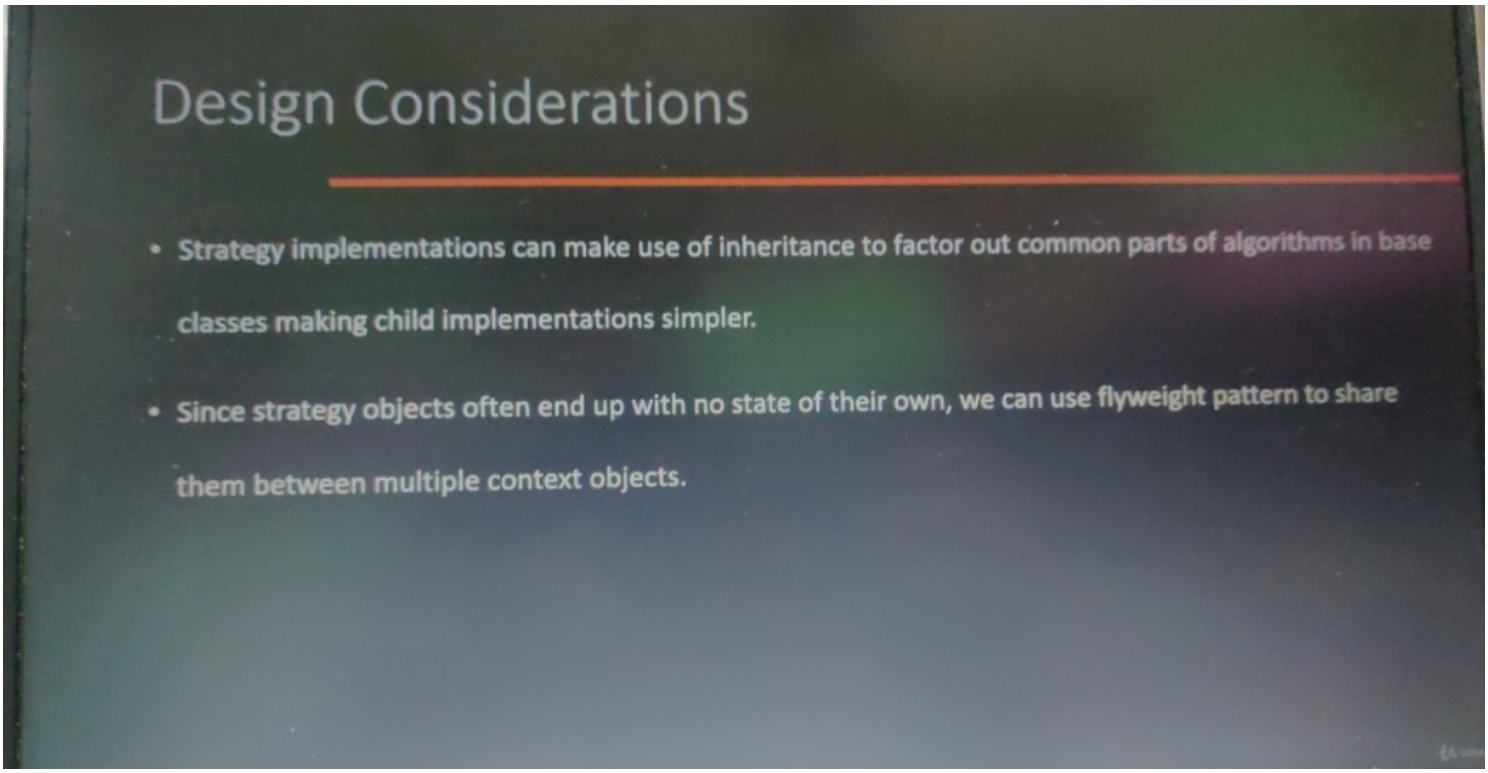
- We start by defining strategy interface which is used by our main/context class. Context class provides strategy with all the data that it needs.
- We provide implementations for various algorithms by implementing strategy interface a class per algorithm.
- Our context class provides a way to configure it with one of the strategy implementations. Client code will create context with one of the strategy object.

Example: UML


class StrategyEx
User
Client
«context»
PrintToFileService
«strategy»
OrderPrinter
DetailPrinter
SummaryPrinter

Implementation Considerations

- We can implement our context in a way where strategy object is optional. This makes context usable for client codes who do not want to deal with concrete strategy objects.
- Strategy objects should be given all data they need as arguments to its method. If number of arguments are high then we can pass strategy an interface reference which it queries for data. Context object can implement this interface and pass itself to strategy.
- Strategies typically end up being stateless objects making them perfect candidates for sharing between context objects.


Design Considerations
• Strategy implementations can make use of inheritance to factor out common parts of algorithms in base classes making child implementations simpler.
• Since strategy objects often end up with no state of their own, we can use flyweight pattern to share them between multiple context objects.

Examples of Strategy pattern

- The java.util.Comparator is a great example of strategy pattern. We can create multiple implementations of comparator, each using a different algorithm to perform comparison and supply those to various sort methods.

```
class User {
    private String name;

    private int age;

    public User(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
```

```
class SortByAge implements Comparator<User> {
    @Override
    public int compare(User o1, User o2) {
        return o1.getAge() - o2.getAge();
    }
}
```

```
class SortByName implements Comparator<User> {
    @Override
    public int compare(User o1, User o2) {
        return o1.getName().compareToIgnoreCase(o2.get
    }
}
```

Examples of Strategy pattern

```
List<User> list = new ArrayList<>();
list.add(new User("Nancy", 16));
list.add(new User("Dustin", 12));
list.add(new User("Steve", 17));
list.add(new User("Mike", 12));
list.add(new User("Max", 13));

list.sort(new SortByAge());

list.sort(new SortByName());
```

- Another example of strategy pattern is the ImplicitNamingStrategy & PhysicalNamingStrategy contracts in Hibernate. Implementations of these classes are used when mapping an Entity to database tables. These classes tell hibernate which table to use & which columns to use.

Compare & Contrast with State

Strategy

- We create a class per algorithm.
- Strategy objects do not need to know about each other.

State

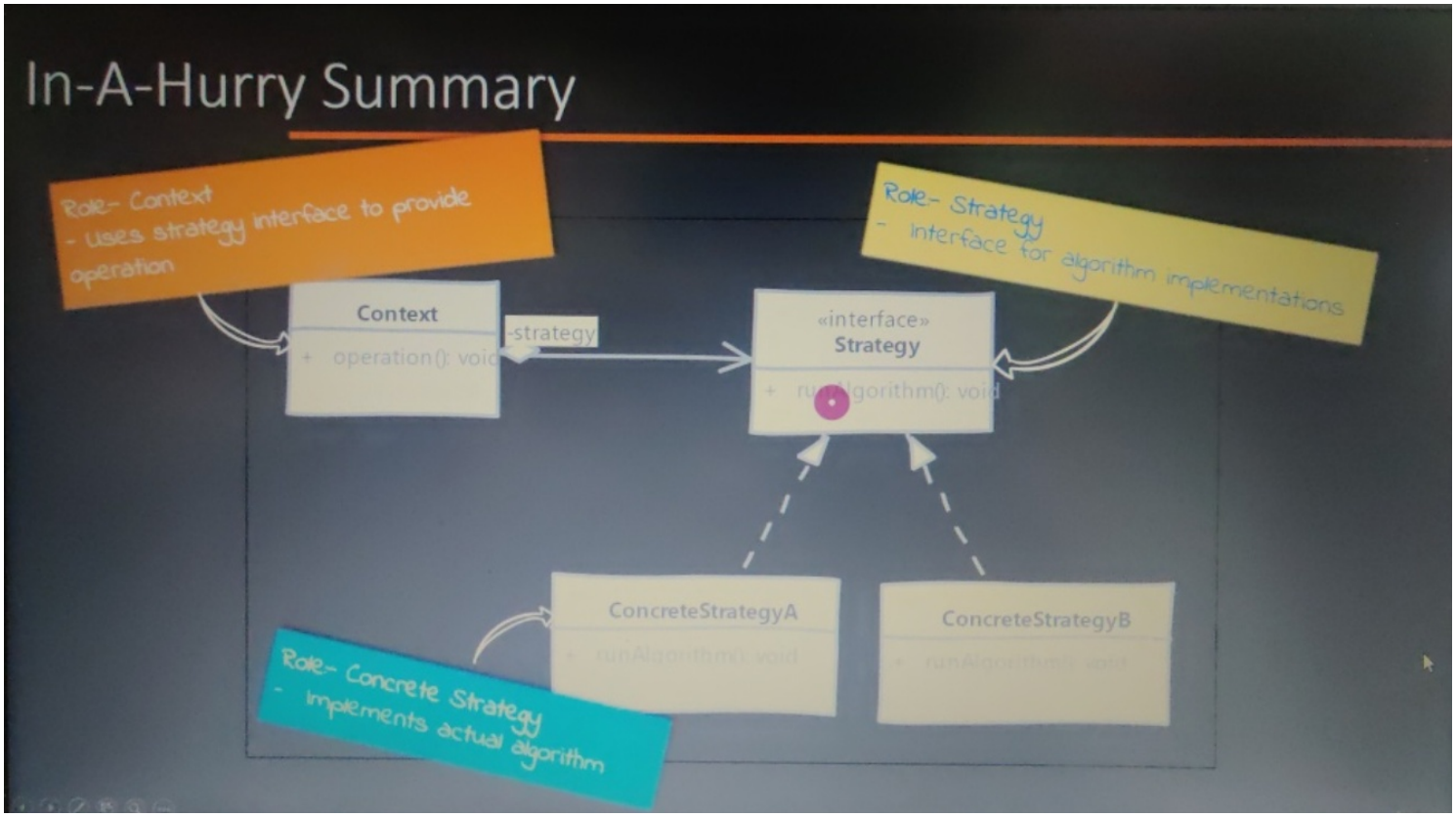
- In state pattern we have a class per state.
- If states are responsible for triggering state transitions then they have to know about at least next state.

Pitfalls

- Since client code configures context object with appropriate strategy object, clients know about all implementations of strategy. Introducing new algorithm means changing client code as well.

In-A-Hurry Summary

- Strategy pattern allows us to encapsulate algorithms in separate classes. The class using these algorithms (called context) can now be configured with desired implementation of an algorithm.
- It is typically the responsibility of client code which is using our context object to configure it.
- Strategy objects are given all data they need by the context object. We can pass data either in form of arguments or pass on context object itself.
- Strategy objects typically end up being stateless making them great candidates for flyweight pattern.
- Client code ends up knowing about all implementations of strategy since it has to create their objects.


In-A-Hurry Summary
Role- Context
- Uses strategy interface to provide operation
Role- Strategy
- Interface for algorithm implementations
Context
+ operation(): void
-strategy
«interface»
Strategy
+ runAlgorithm(): void
ConcreteStrategyA
+ runAlgorithm(): void
ConcreteStrategyB
+ runAlgorithm(): void
Role- Concrete Strategy
- Implements actual algorithm