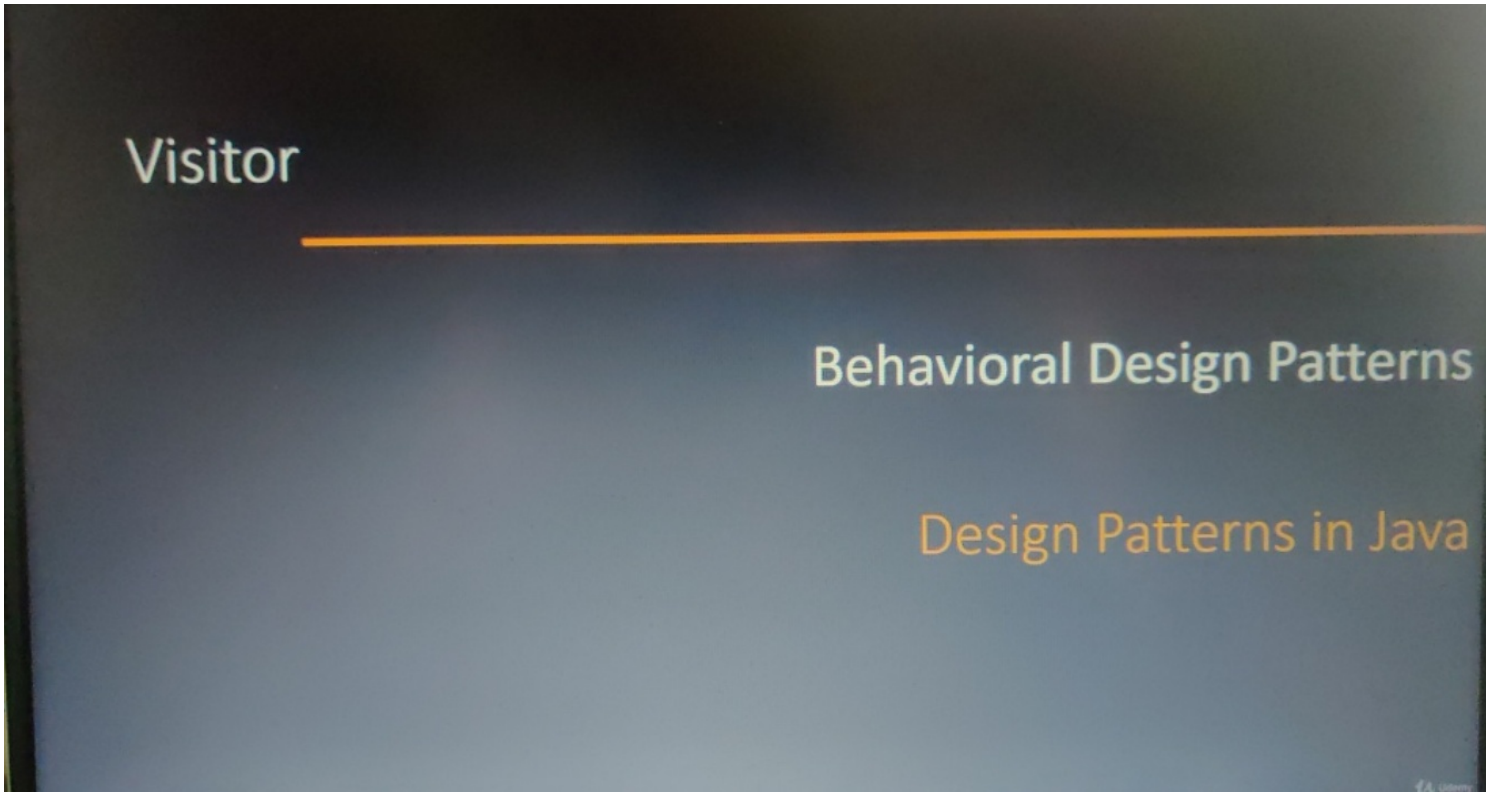

Visitor
Behavioral Design Patterns
Design Patterns in Java

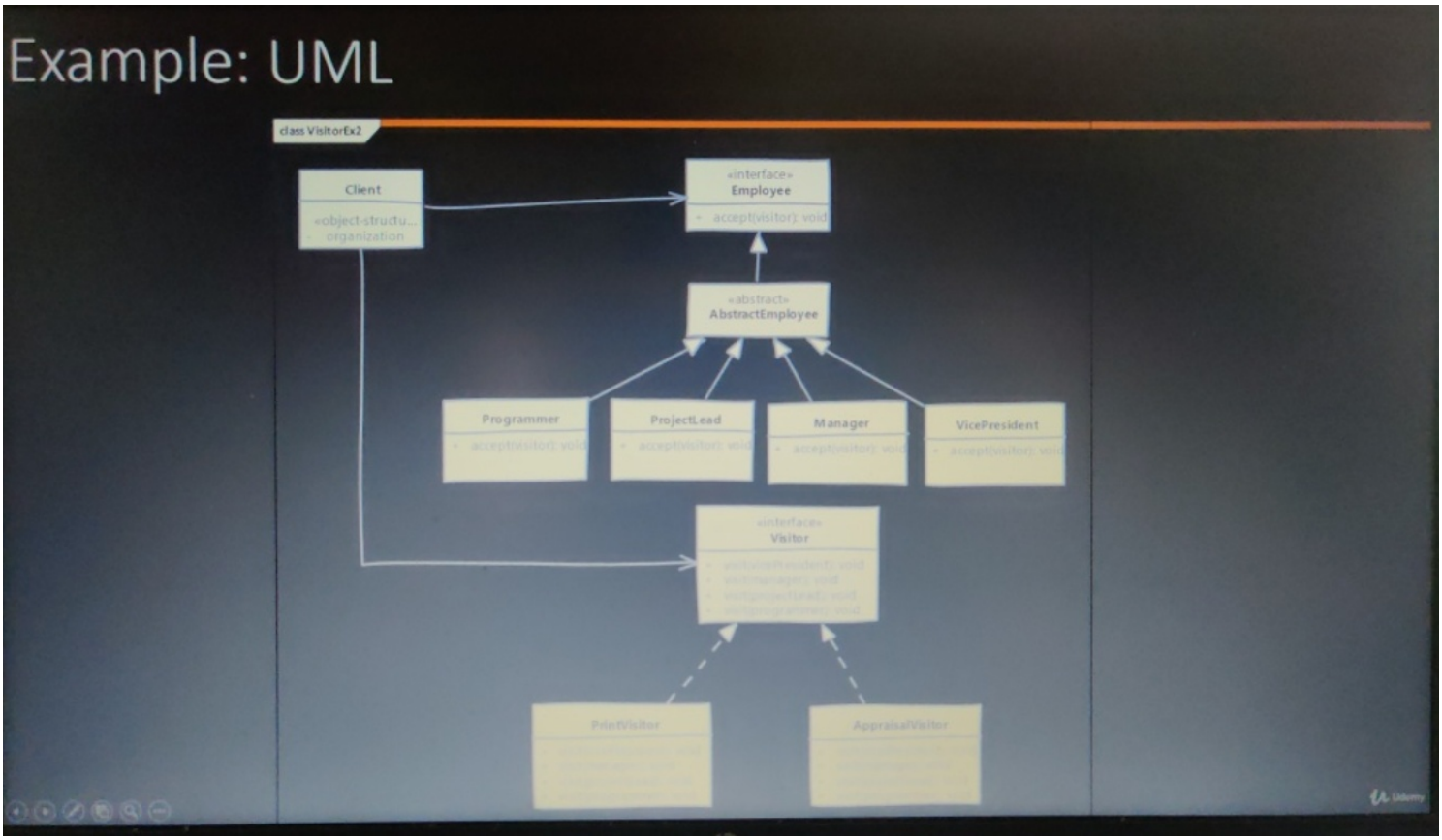
What is a Visitor Pattern?

- Visitor pattern allows us to define new operations that can be performed on an object without changing the class definition of the object.
- Think of this pattern as an object ("visitor") that visits all nodes in an object structure. Each time our visitor visits a particular object from the object structure, that object calls a specific method on visitor, passing itself as an argument.
- Each time we need a new operation we create a subclass of visitor, implement the operation in that class and visit the object structure.
- Objects themselves only implement an "accept" visit where the visitor is pass as an argument. Objects know about the method in visitor created specifically for it and invoke that method inside the accept method.


UML
class Visitor
Client
«interface»
Visitor
+ visitConcreteElementA(concreteElementA)
+ visitConcreteElementB(concreteElementB)
ConcreteVisitorA
ConcreteVisitorB
ObjectStructure
«interface»
Element
ConcreteElementA
ConcreteElementB
Role- Visitor
- Defines visit operation for each concrete element class
Role- Concrete Visitor
- Implements each operation
Role- Object Structure
- Can enumerate elements
Role- Element
- Defines accept operation
Role- Concrete Element
- Implements accept and calls visitor's method which is defined for this class

Implement Visitor Pattern

- We create visitor interface by defining "visit" methods for each class we want to support.
- The classes who want functionalities provided by visitor define "accept" method which accepts a visitor. These methods are defined using the visitor interface as parameter type so that we can pass any class implementing the visitor to these methods.
- In the accept method implementation we'll call a method on visitor which is defined specifically for that class.
- Next we implement the visitor interface in one or more classes. Each implementation provides a specific functionality for interested classes. If want another feature we create new implementation of visitor.


Example: UML
class VisitorEx2
Client
«object-structu...
- organization
«interface»
Employee
+ acceptVisitor(): void
«abstract»
AbstractEmployee
Programmer
ProjectLead
Manager
VicePresident
«interface»
Visitor
PrintVisitor
AppraisalVisitor

Implementation Considerations

- Visitor can work with objects of classes which do not have a common parent. So having a common interface for those classes is optional. However the code which passes our visitor to these objects must be aware of these individual classes.
- Often visitors need access to internal state of objects to carry out their work. So we may have to expose the state using getters/setters.

Design Considerations

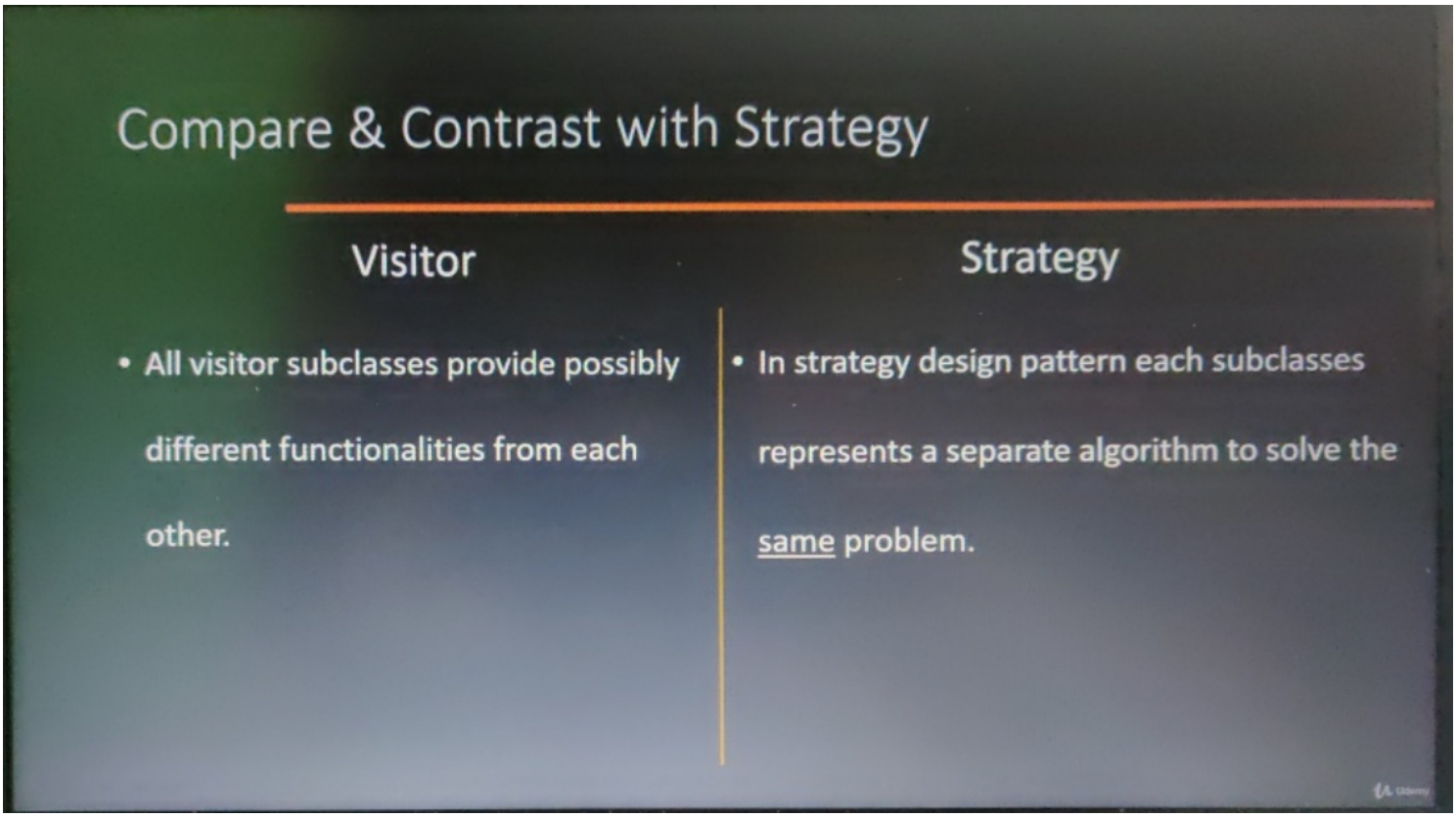
- One effect of this pattern is that related functionality is grouped in a single visitor class instead of spread across multiple classes. So adding new functionality is as simple as adding a new visitor class.
- Visitors can also accumulate state. So along with behavior we can also have state per object in our visitor. We don't have to add new state to objects for behavior defined in visitor.
- Visitor can be used to add new functionality to object structure implemented using composite or can be used for doing interpretation in interpreter design pattern.

Examples of Visitor Pattern

- The dom4j library used for parsing XML has interface org.dom4j.Visitor & implementation org.dom4j.VisitorSupport which are examples of visitor. By implementing this visitor we can process each node in an XML tree.
- Another example of visitor pattern is the java.nio.file.FileVisitor & its implementation SimpleFileVisitor.

```
class DeleteVisitor extends SimpleFileVisitor<Path> {
    @Override
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs)
        throws IOException
    {
        Files.delete(file);
        return FileVisitResult.CONTINUE;
    }
    @Override
    public FileVisitResult postVisitDirectory(Path dir, IOException e)
        throws IOException
    {
        if (e == null) {
            Files.delete(dir);
            return FileVisitResult.CONTINUE;
        } else {
            // directory iteration failed
            throw e;
        }
    }
};
```

Modified example from Javadoc

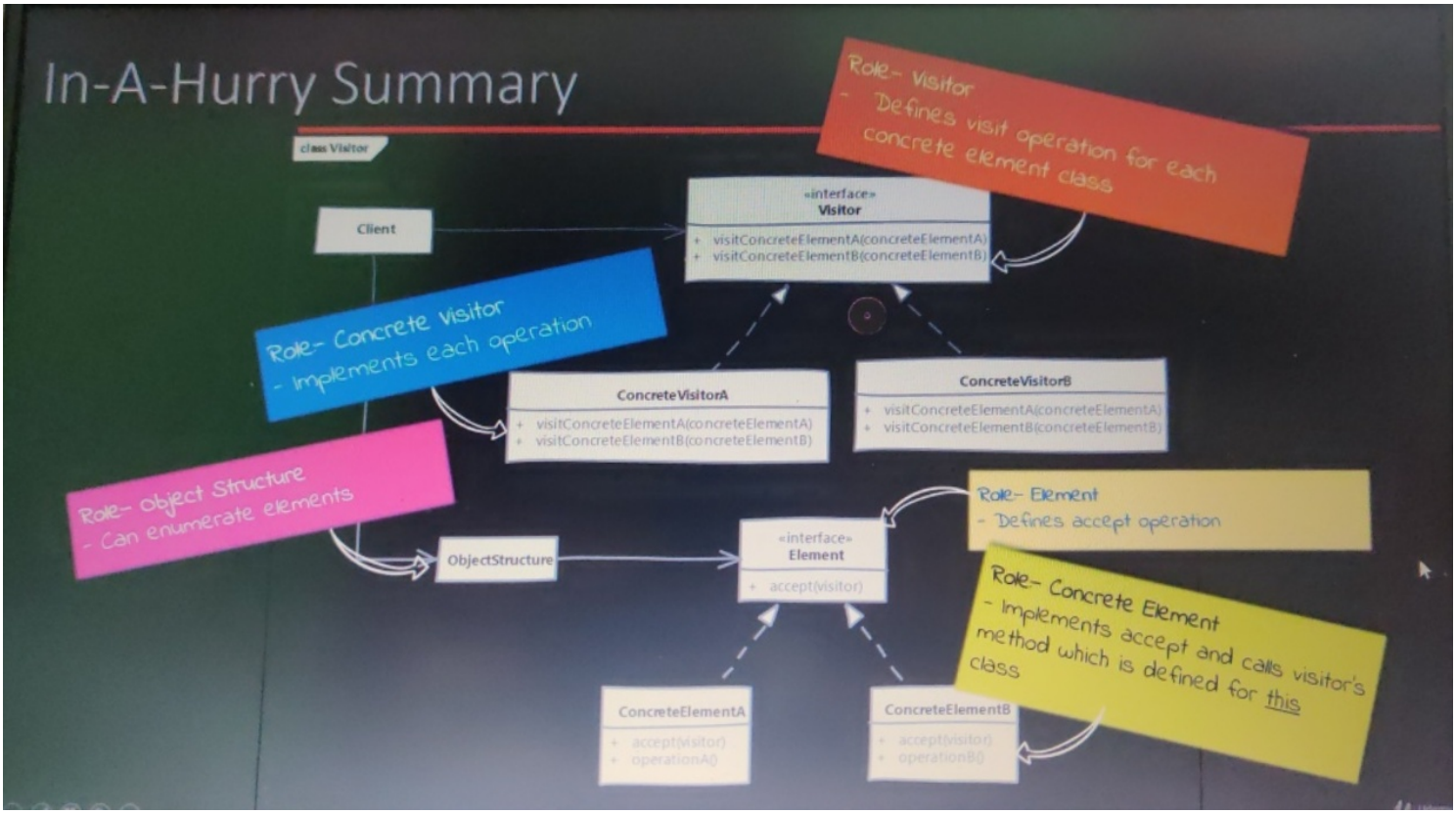

Compare & Contrast with Strategy
Visitor
Strategy
• All visitor subclasses provide possibly different functionalities from each other.
• In strategy design pattern each subclasses represents a separate algorithm to solve the same problem.

Pitfalls

- Often visitors need access to object's state. So we end up exposing a lot of state through getter methods, weakening the encapsulation.
- Supporting a new class in our visitors requires changes to all visitor implementations.
- If the classes themselves change then all visitors have to change as well since they have to work with changed class.
- A little bit confusing to understand and implement.

In-A-Hurry Summary

- Visitor pattern allows to add new operations that work on objects without modifying class definitions of these objects.
- Visitors define class specific methods which work with an object of that class to provide new functionality.
- To use this pattern classes define a simple accept method which gets a reference to a visitor and inside this method, objects class method on visitor which is defined for that specific class.
- Adding a new functionality means creating a new visitor and implementing new functionality in that class instead of modifying each class where this functionality is needed.
- This pattern is often used where we have an object structure and then another class or visitor itself iterates over this structure passing our visitor object to each object.


In-A-Hurry Summary
class Visitor
Client
«interface»
Visitor
+ visitConcreteElementA(concreteElementA)
+ visitConcreteElementB(concreteElementB)
Role- Visitor
- Defines visit operation for each concrete element class
Role- Concrete Visitor
- Implements each operation
ConcreteVisitorA
+ visitConcreteElementA(concreteElementA)
+ visitConcreteElementB(concreteElementB)
ConcreteVisitorB
+ visitConcreteElementA(concreteElementA)
+ visitConcreteElementB(concreteElementB)
Role- Object Structure
- Can enumerate elements
ObjectStructure
«interface»
Element
+ accept(visitor)
Role- Element
- Defines accept operation
Role- Concrete Element
- Implements accept and calls visitor's method which is defined for this class
ConcreteElementA
+ accept(visitor)
+ operationA()
ConcreteElementB
+ accept(visitor)
+ operationB()