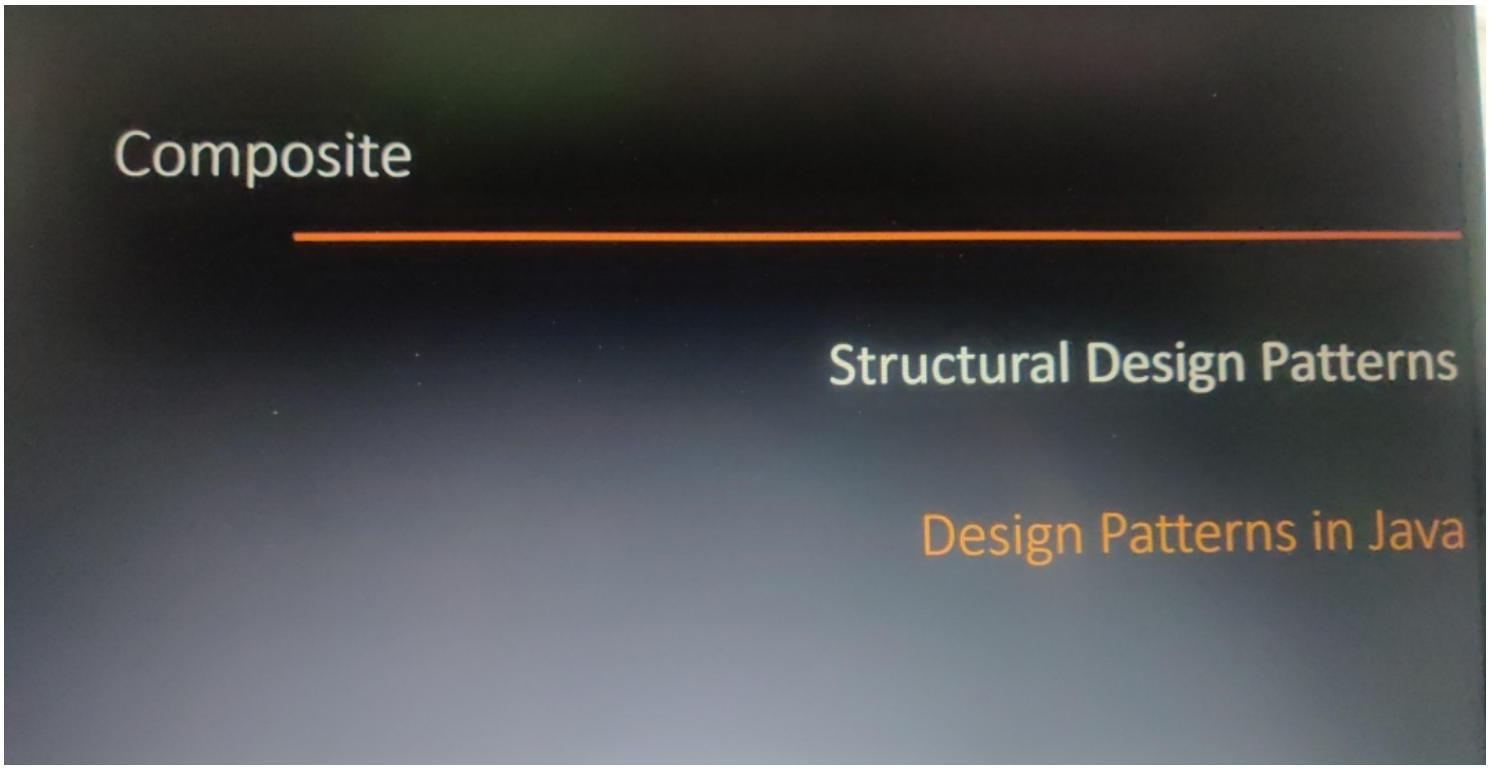
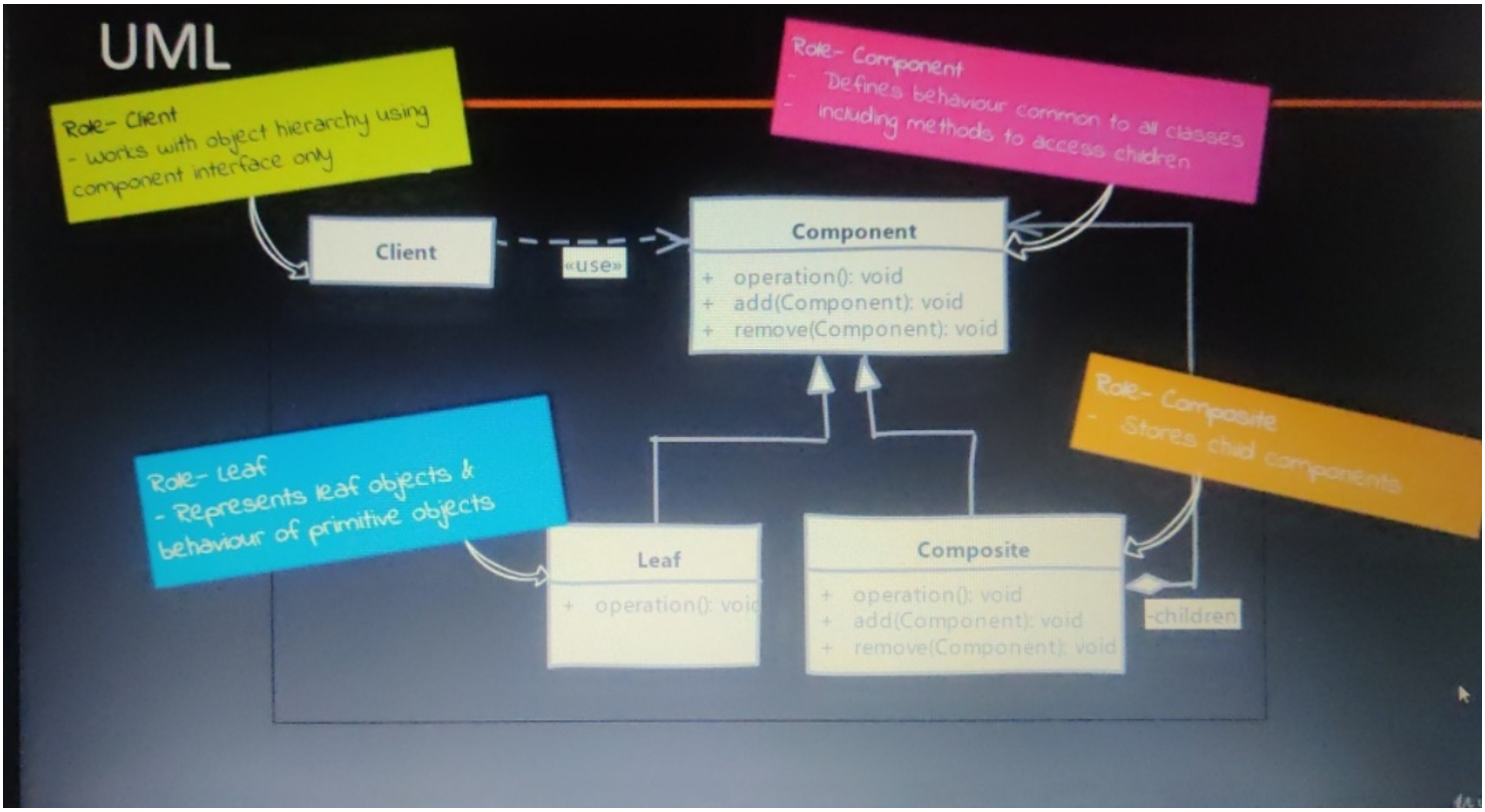

Composite
Structural Design Patterns
Design Patterns in Java

What is Composite?

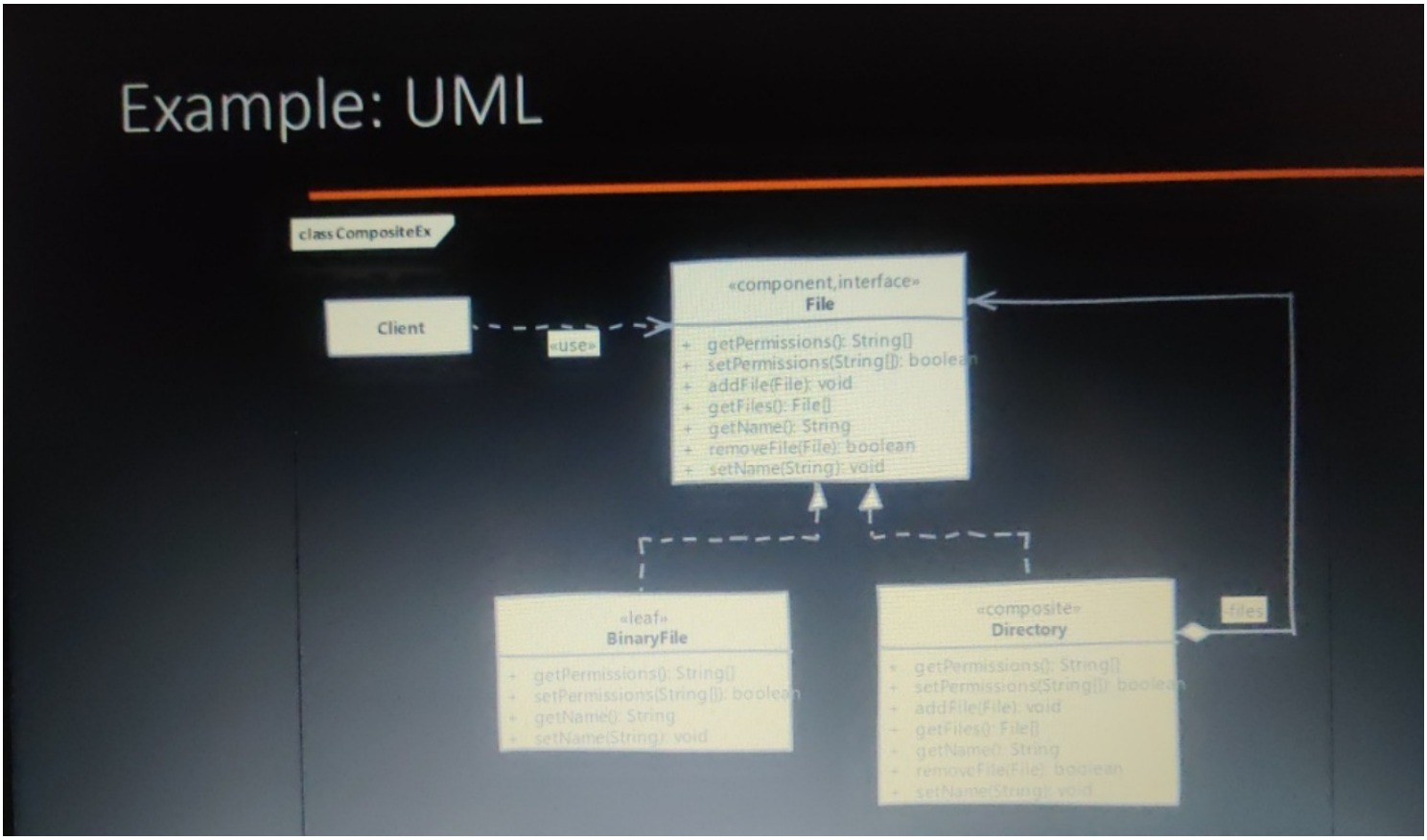
- We have a part-whole relationship or hierarchy of objects and we want to be able to treat all objects in this hierarchy uniformly.
- This is NOT a simple composition concept from object oriented programming but a further enhancement to that principal.
- Think of composite pattern when dealing with tree structure of objects.


UML
Role- Client
- Works with object hierarchy using component interface only
Role- Component
- Defines behaviour common to all classes
- including methods to access children
Client
«use»
Component
+ operation(): void
+ add(Component): void
+ remove(Component): void
Role- Composite
- Stores child components
Role- Leaf
- Represents leaf objects &
behaviour of primitive objects
Leaf
+ operation(): void
Composite
+ operation(): void
+ add(Component): void
+ remove(Component): void
-children

Implement a Composite

- We start by creating an abstract class / interface for Component
  - Component must declare all methods that are applicable to both leaf and composite.
  - We have to choose who defines the children management operations, component or composite.
  - Then we implement the composite. An operation invoked on composite is propagated to all its children
  - In leaf nodes we have to handle the non-applicable operations like add/remove a child if they are defined in component.
- In the end, a composite pattern implementation will allow you to write algorithms without worrying abou whether node is leaf or composite.

Example: UML


class CompositeEx
Client
«use»
«component,interface»
File
+ getPermissions(): String[]
+ setPermissions(String[]): boolean
+ addFile(File): void
+ getFiles(): File[]
+ getName(): String
+ removeFile(File): boolean
+ setName(String): void
«leaf»
BinaryFile
+ getPermissions(): String[]
+ setPermissions(String[]): boolean
+ getName(): String
+ setName(String): void
«composite»
Directory
+ getPermissions(): String[]
+ setPermissions(String[]): boolean
+ addFile(File): void
+ getFiles(): File[]
+ getName(): String
+ removeFile(File): boolean
+ setName(String): void
files

Implementation Considerations

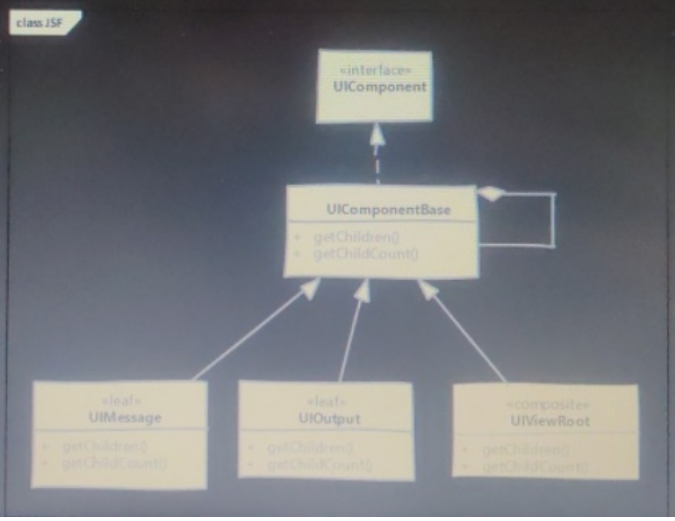
- You can provide a method to access parent of a node. This will simplify traversal of the entire tree.
- You can define the collection field to maintain children in base component instead of composite but again that field has no use in leaf class.
- If leaf objects can be repeated in the hierarchy then shared leaf nodes can be used to save memory and initialization costs. But again the number of nodes is major deciding factor as using a cache for small total number of nodes may cost more.

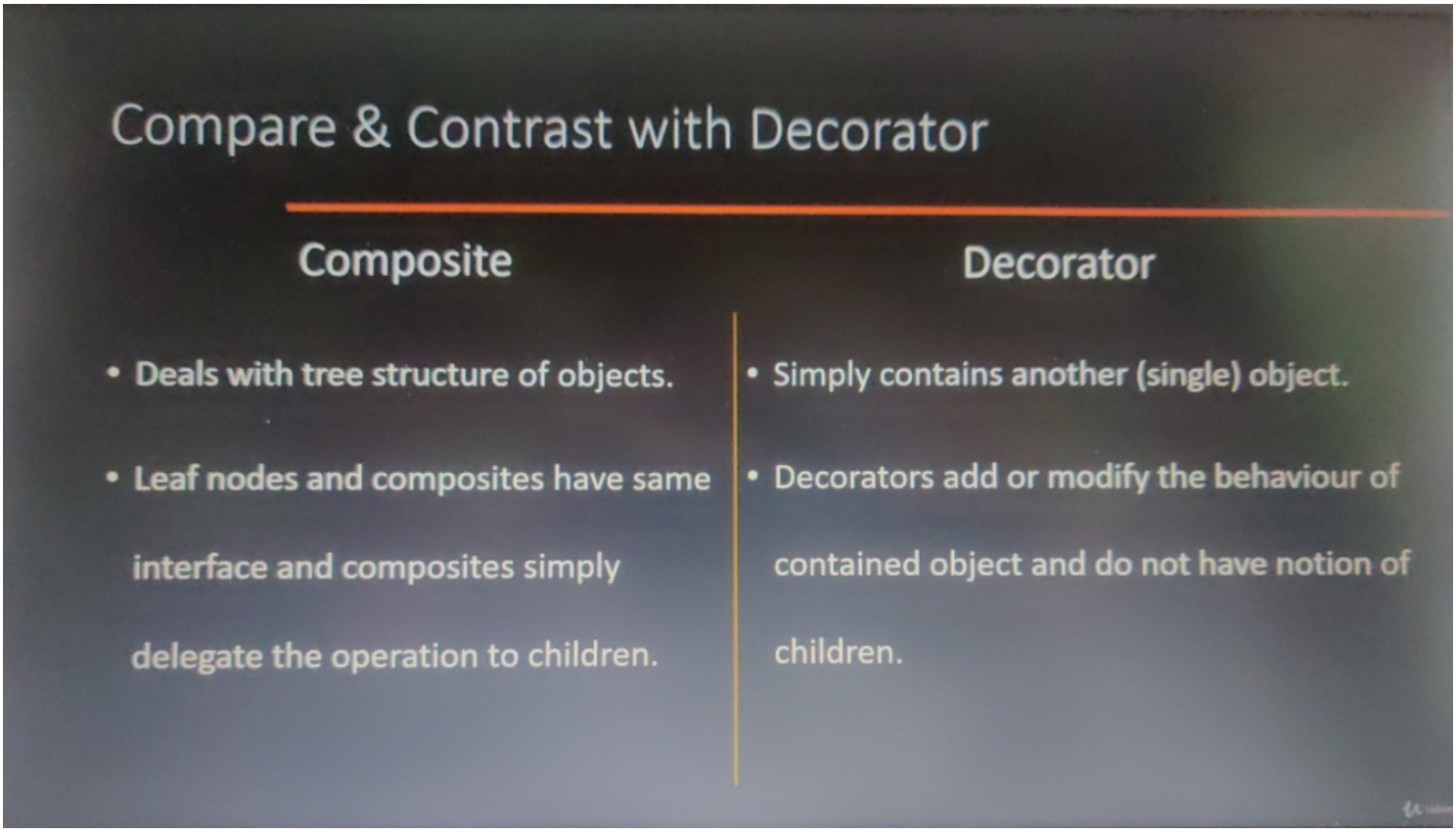
Design Considerations

- Decision needs to be made about where child management operations are defined. Defining them on component provides transparency but leaf nodes are forced to implement those methods. Defining them on composite is safer but client needs to be made aware of composite.
- Overall goal of design should be to make client code easier to implement when using composite. This is possible if client code can work with component interface only and doesn't need to worry about leaf-composite distinction.

Example of a Composite

- Composite is used in many UI frameworks, since it easily allows to represent a tree of UI controls.
- In JSF we have UIViewRoot class which acts as composite. Other UIComponent implementations like UIOutput, UIMessage act as leaf nodes.



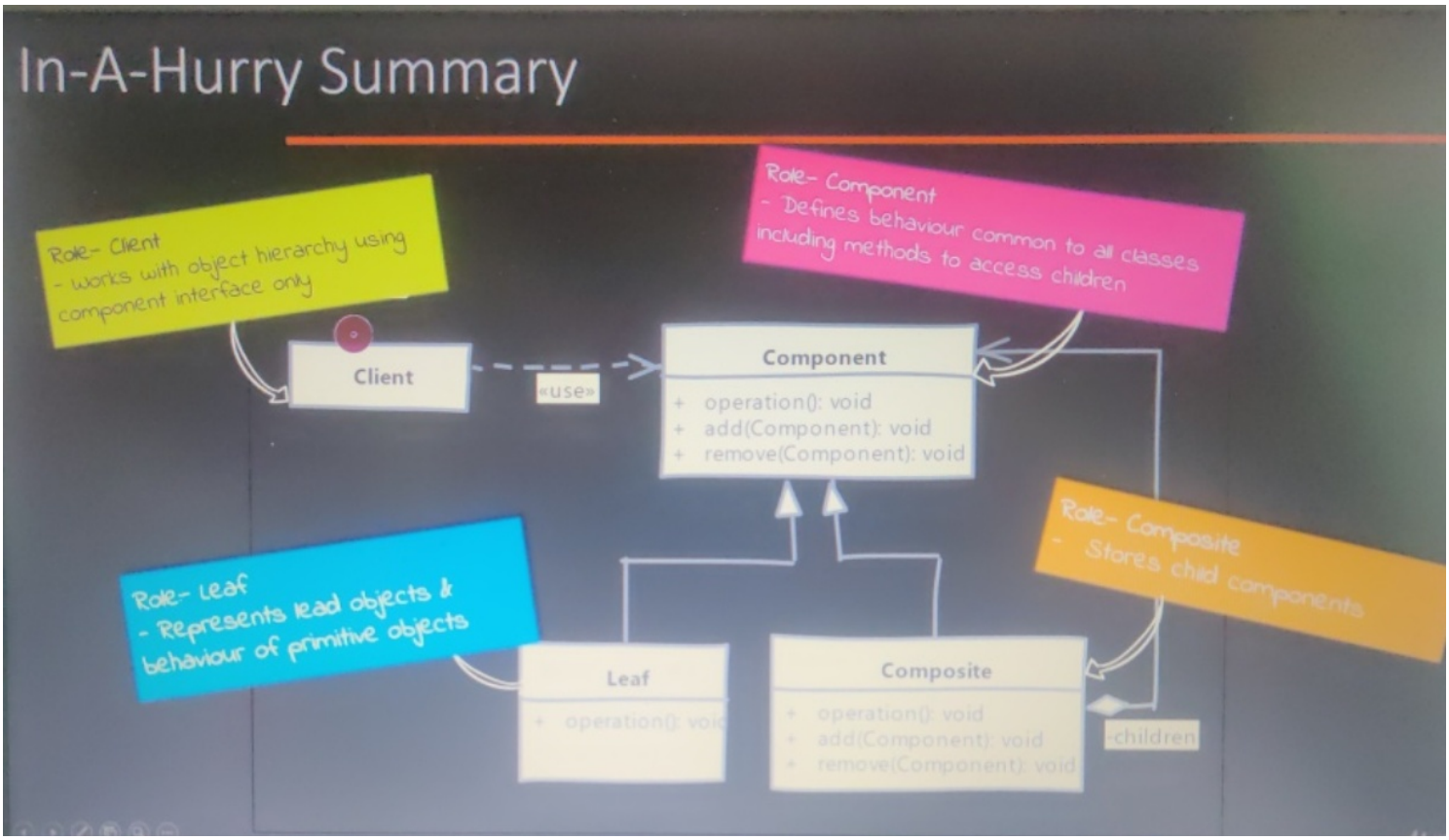

Compare & Contrast with Decorator
Composite
Deals with tree structure of objects.
Leaf nodes and composites have same interface and composites simply delegate the operation to children.
Decorator
Simply contains another (single) object.
Decorators add or modify the behaviour of contained object and do not have notion of children.

Pitfalls

- Difficult to restrict what is added to hierarchy. If multiple types of leaf nodes are present in system then client code ends up doing runtime checks to ensure the operation is available on a node.
- Creating the original hierarchy can still be complex implementation especially if you are using caching to reuse nodes and number of nodes are quite high.

In-A-Hurry Summary

- We have a parent-child or whole-part relation between objects. We can use composite pattern to simplify dealing with such object arrangements.
- Goal of composite pattern is to simplify the client code by allowing it to treat the composites and leaf nodes in same way.
- Composites will delegate the operations to its children while leaf nodes implement the functionality.
- You have to decide which methods the base component will define. Adding all methods here will allow client to treat all nodes same. But it may force classes to implement behaviour which they don't have.


In-A-Hurry Summary
Role- Client
- Works with object hierarchy using component interface only
Role- Component
- Defines behaviour common to all classes including methods to access children
Client
«use»
Component
+ operation(): void
+ add(Component): void
+ remove(Component): void
Role- Leaf
- Represents lead objects & behaviour of primitive objects
Role- Composite
- Stores child components
Leaf
+ operation(): void
Composite
+ operation(): void
+ add(Component): void
+ remove(Component): void
-children