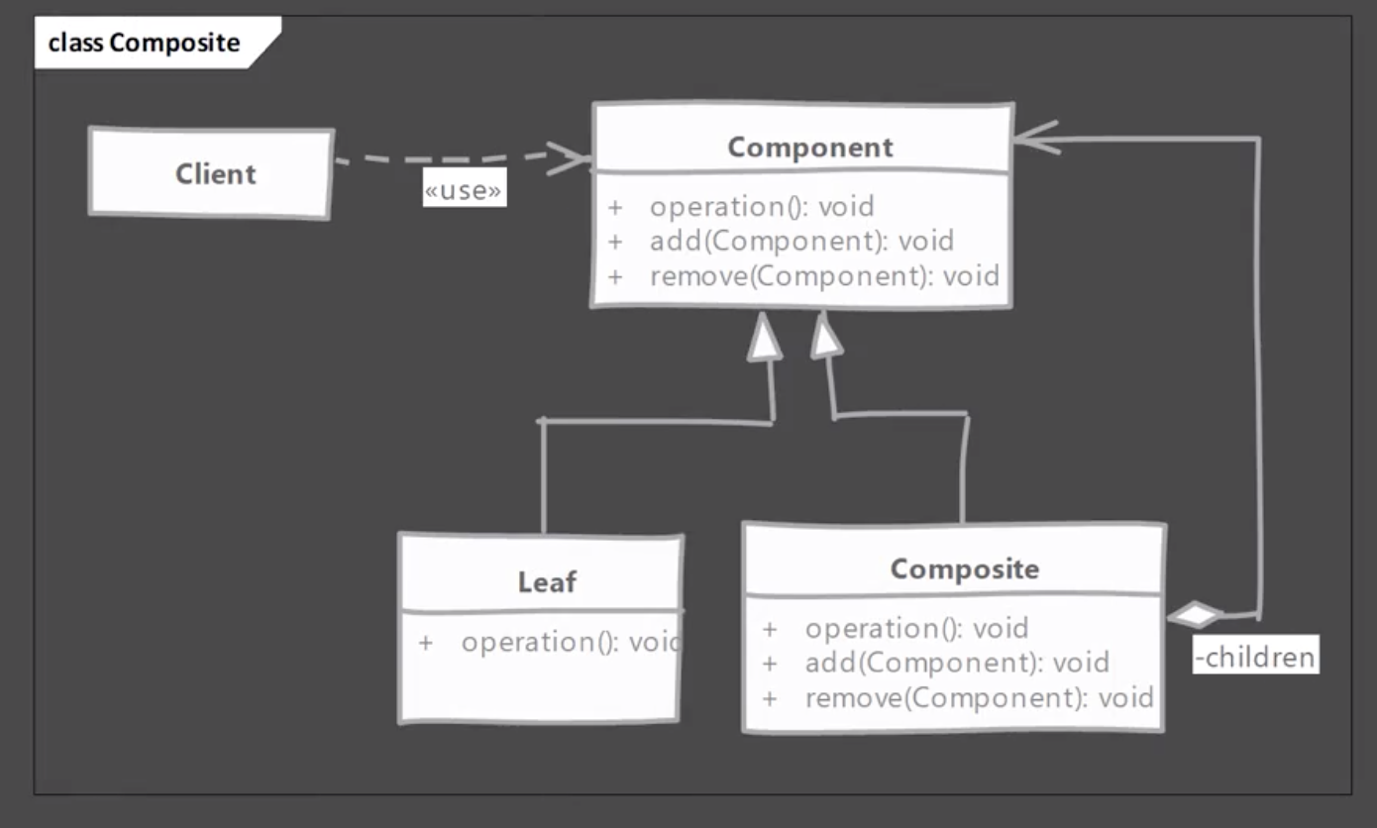
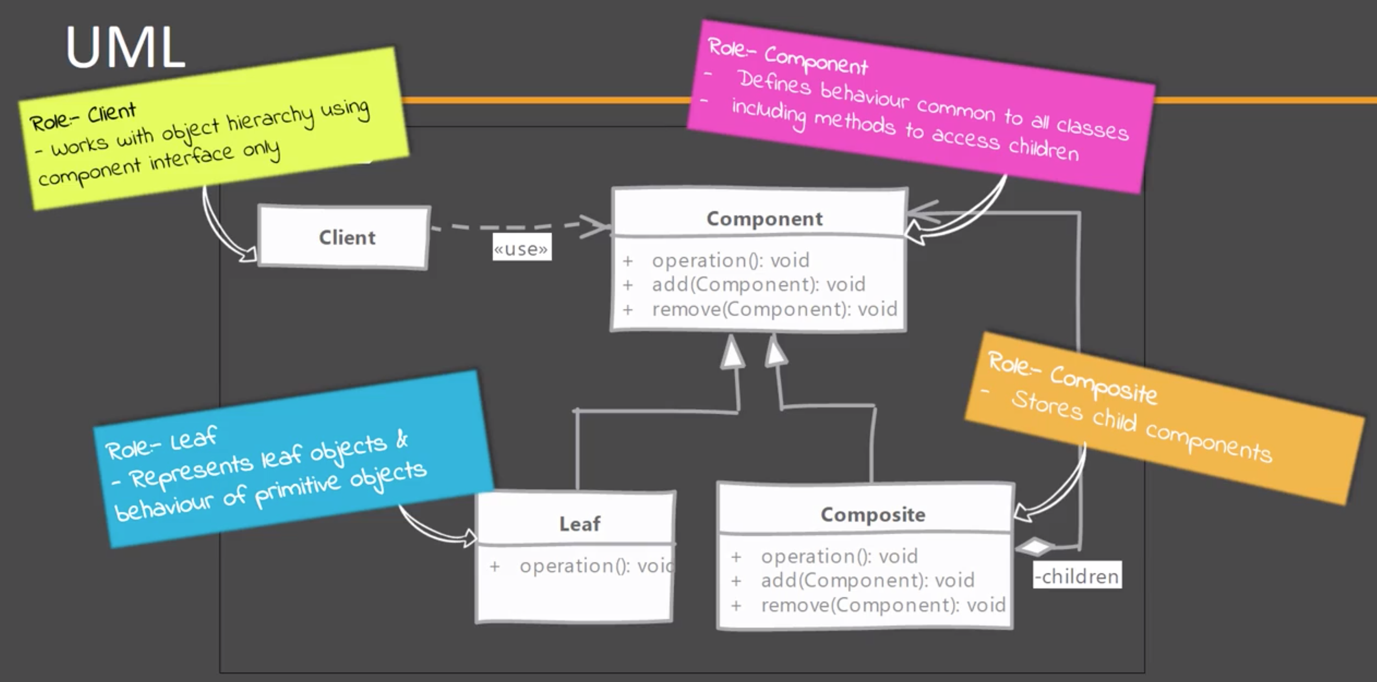
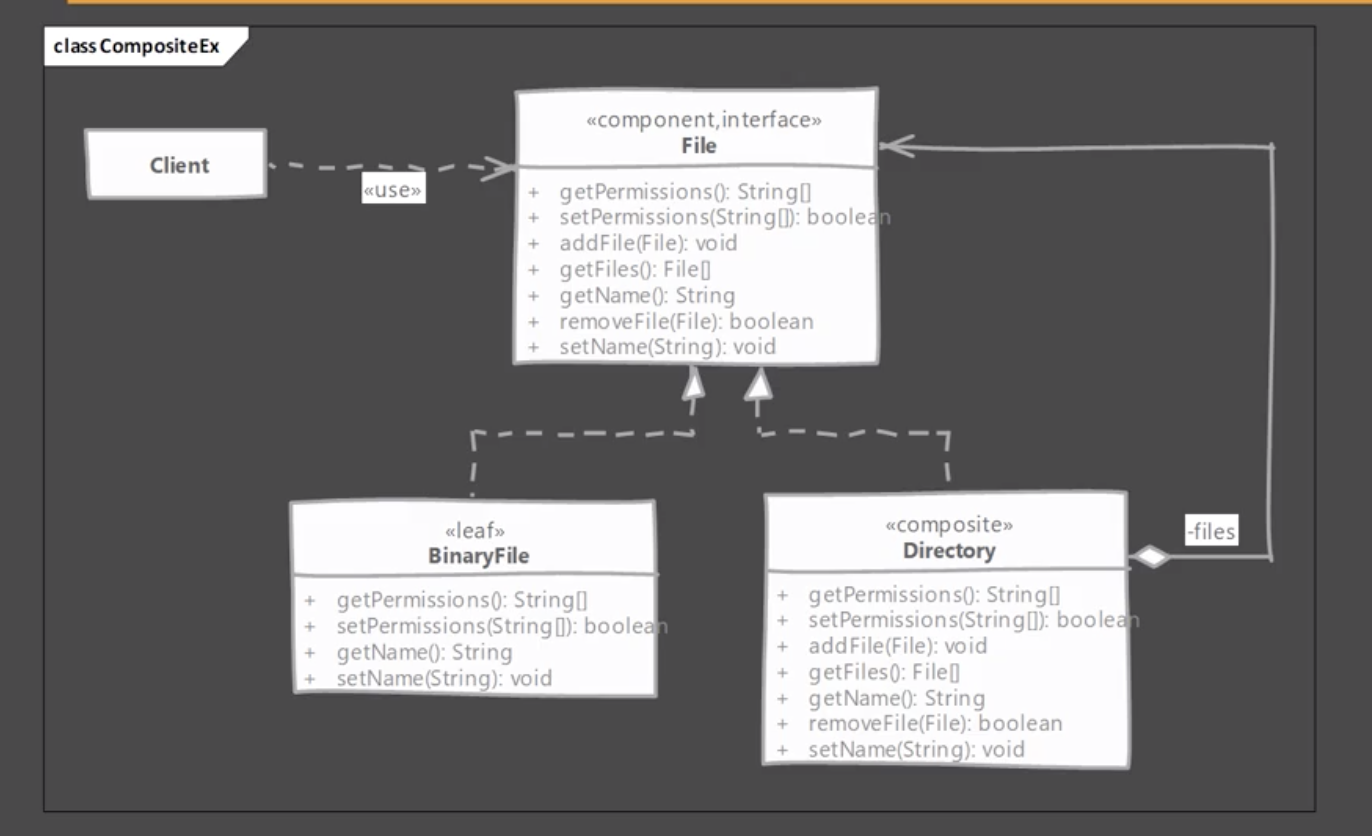
**Composite Pattern**

* **Introduction:**
  + 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:**

****

****

* **Implementation Steps:**
  + We start by creating an abstract class / interface for Component.
    - Component must declare all methods that are applicable to 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 node 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 about whether node is leaf or composite.
* **Example UML:**

****

* **Implementation & Design Consideration:**
  + 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 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 Consideration:**

* + Decisions needs to be made about where child management operations are defined. Defining on 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:**
  + 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 acts as leaf nodes.

**A screenshot of a cell phone

Description automatically generated**

* **Comparison with Decorator:**

|  |  |
| --- | --- |
| **Composite** | **Decorator** |
| Deals with tree structure of objects. | Simply contains another (single) object. |
| Leaf nodes and composites have same interface and composites simply delegate the operation to children. | 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 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.
* **Quiz:**

**A screenshot of a social media post

Description automatically generated**

**A screenshot of a social media post

Description automatically generated**

**A screenshot of a social media post

Description automatically generated**

**A screenshot of a social media post

Description automatically generated**