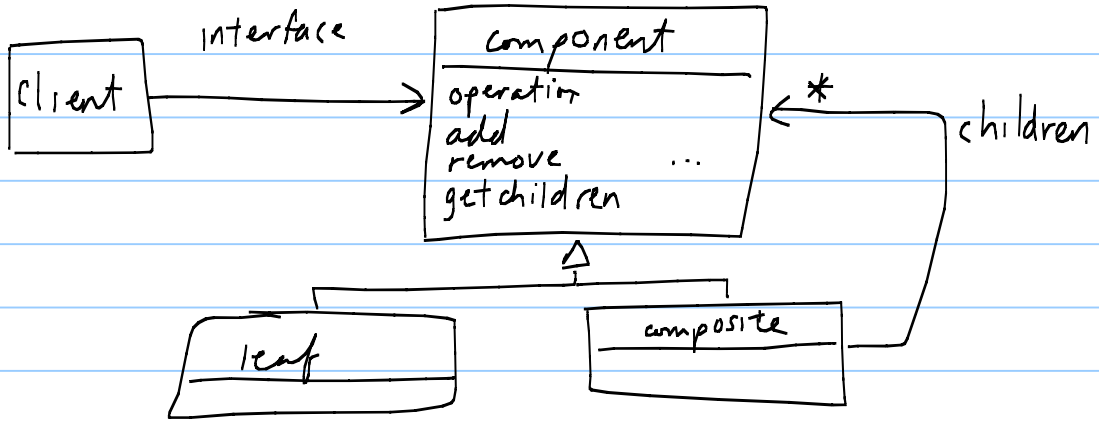
Comp 304 REEHAN SHAIKH

Note Title 3/20/2006



CONSEQUENCES:

+ Client code can be given -primitive -composite target
  ⟹ simple client code
+ easy to add new component sub-class
  (no need to change client code) ++ Reuse
− can be overly general :: no uniform interface
  ie: if we have code that only belongs in leaf, we must put it in Component so client can access it. But then leaf specific code goes into composite. Not good ... too general. So, we move code into leaf but client needs to interact w/ leaf and not component, not good.

· Implementation issues

1) Parent references. co: Composite

```
for all ch in getChildren()
    ch.parent = co
```

2) Problem: if a child has multiple parents.

3) Maximizing the component interface:
+ Clients need not be aware if they're dealing w/ -primitive -composite targets.
- BAD class design. Meaningless operations! ie: getchildren() for leafs!

4) Child management
Add() & remove() methods should go where?
Component level ~ transparency → client only deals w/ interface
Composite level ~ safety → ∅ exceptions can be thrown since leaf doesn't have invalid methods. → RUNTIME

Also @ compile time, if we have a leaf.add() call, we can statically check this.

5) Component (vs composite) keeps reference to children.
⇒ Memory penalty since leaf will also have a list for children and if we have many leafs, waste of space since leafs don't have children

6) Child ordering ie: if we draw shapes, we need to know which shape is above other shapes. So, we can just store the children in order.
∴ we have to implement a data structure

7) Caching children lookup: Each composite caches it's # of children. So, if a new composite is added, we can easily compute the total # of children.
Memory vs. speed.

*) Who should delete?
Sending delete to a component, should we cascade delete or not?

Observer Pattern (Behaviour)

Intent:) many dependencies btwn obj
if an obj. changes, we let our dependents know

AKA: PUBLISH - SUBSCRIBE, OBSERVER, DEPENDENTS