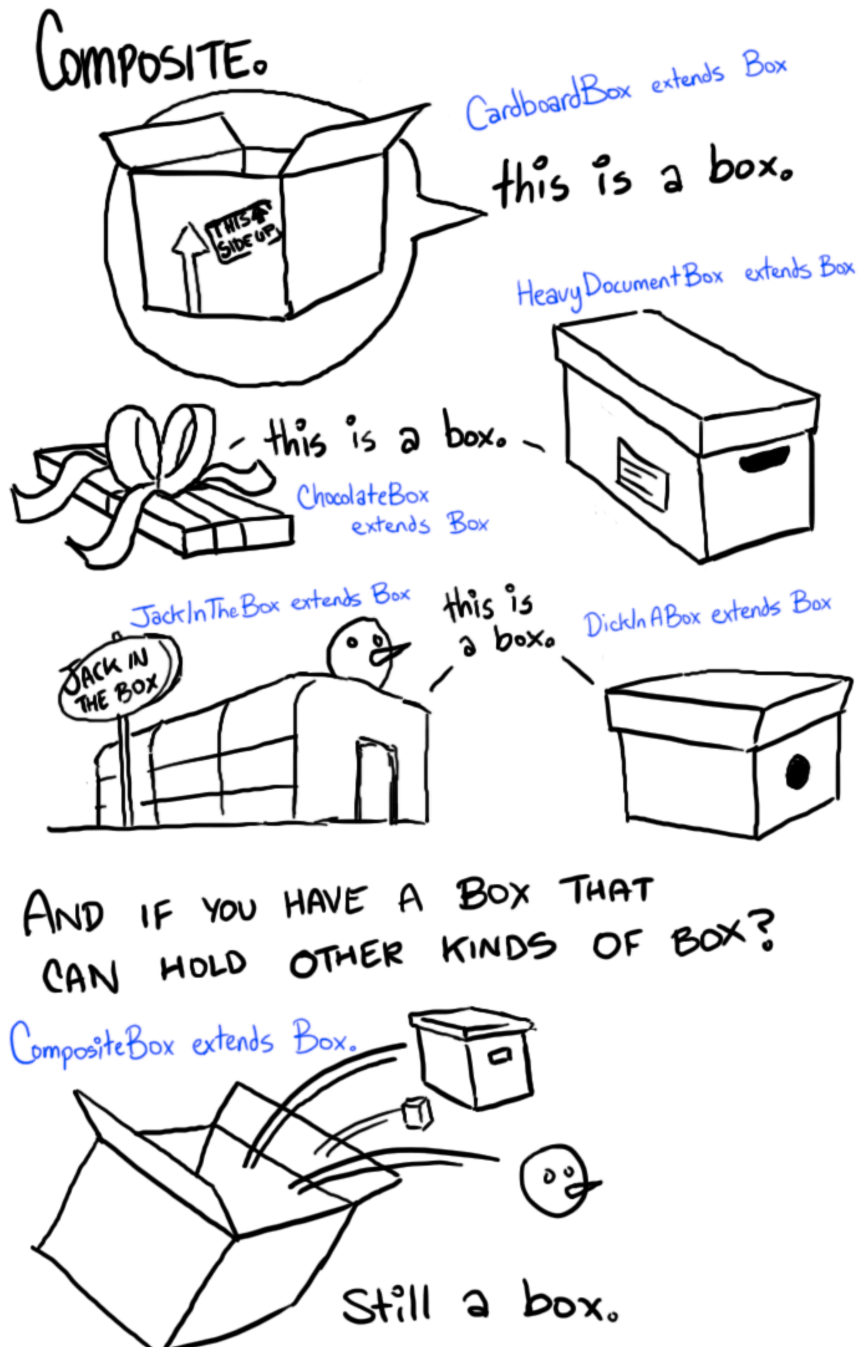
Composite Pattern

CSCI-4448 - Boese


COMPOSITE.
CardboardBox extends Box
this is a box.
HeavyDocumentBox extends Box
THIS SIDE UP
this is a box.
ChocolateBox extends Box
JackInTheBox extends Box
this is a box.
DickInABox extends Box
JACK IN THE BOX
AND IF YOU HAVE A BOX THAT CAN HOLD OTHER KINDS OF BOX?
CompositeBox extends Box.
Still a box.

Objectives

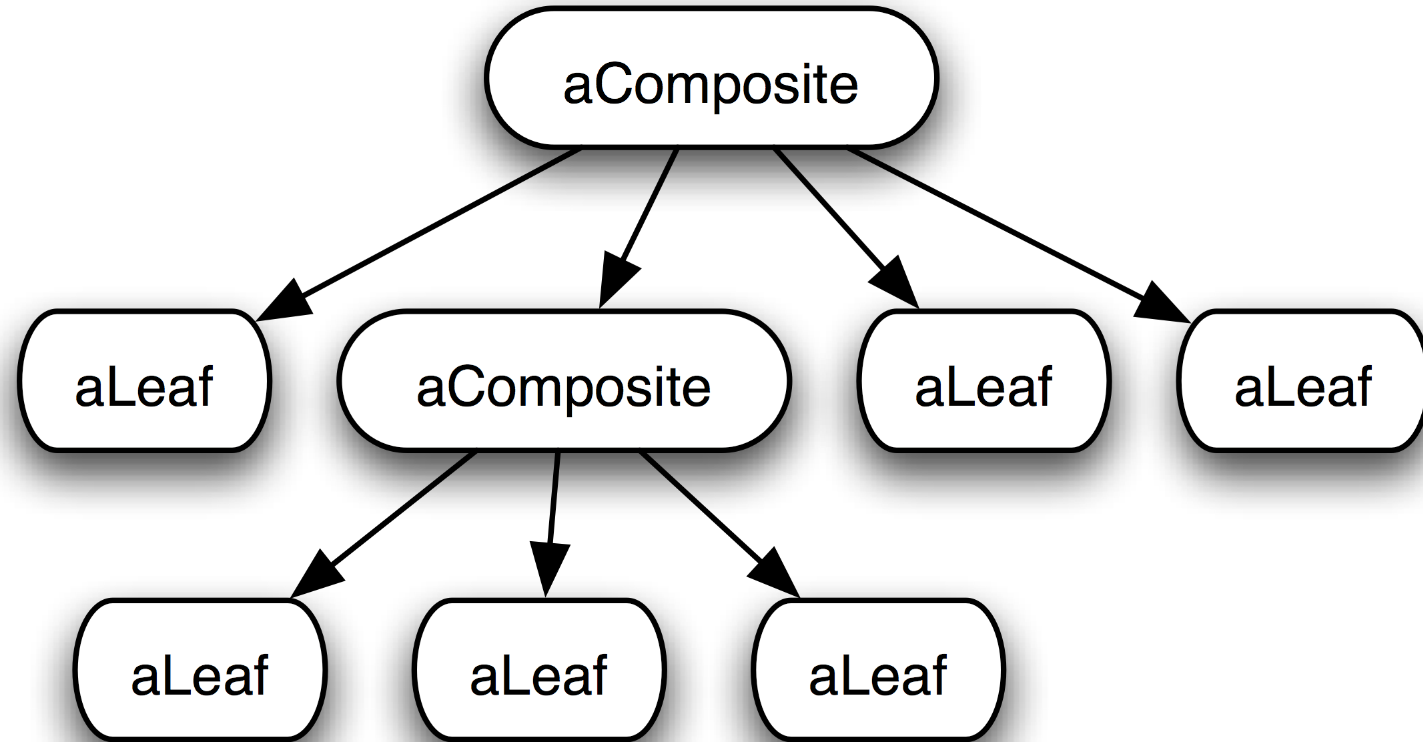
- Problem
- Definition
- Why
- How
- Composite Examples
- Design Considerations

Problem

Why

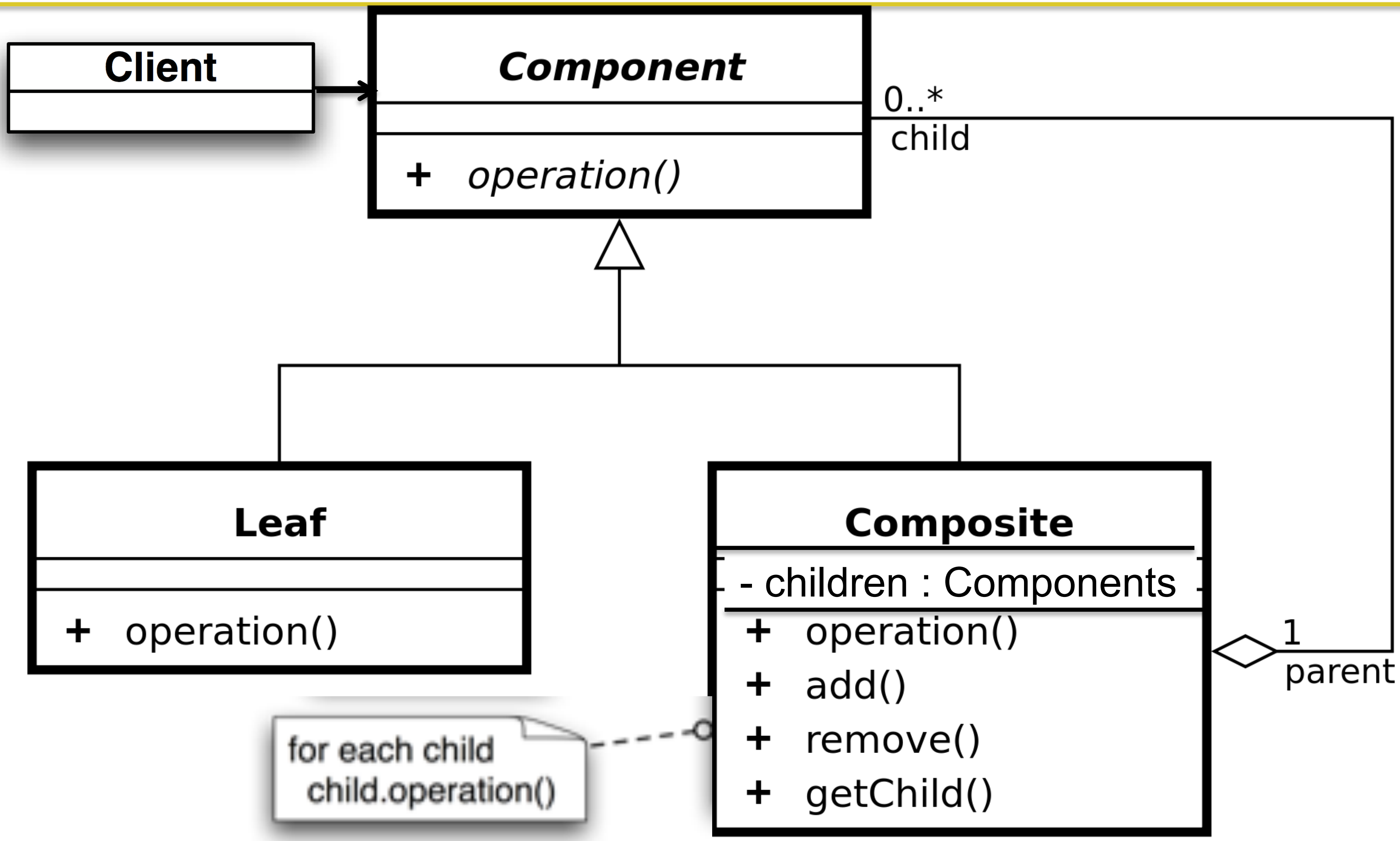
- Some systems can be represented as *whole-part hierarchies*
  - 3D scene graphs, GUIs, robotics, programs…
- Some parts are ***primitives***
  - polygons, lines, actuators, literals…
- Some parts are ***compositions***
  - model, rectangle, arm, expression…
- Clients should be able to view these two sets of components as the same thing
  - Treat a *primitive* and *composition* as if they were in the same class
  - Otherwise, *lots* of run-time checking for what it is working with!

Composite



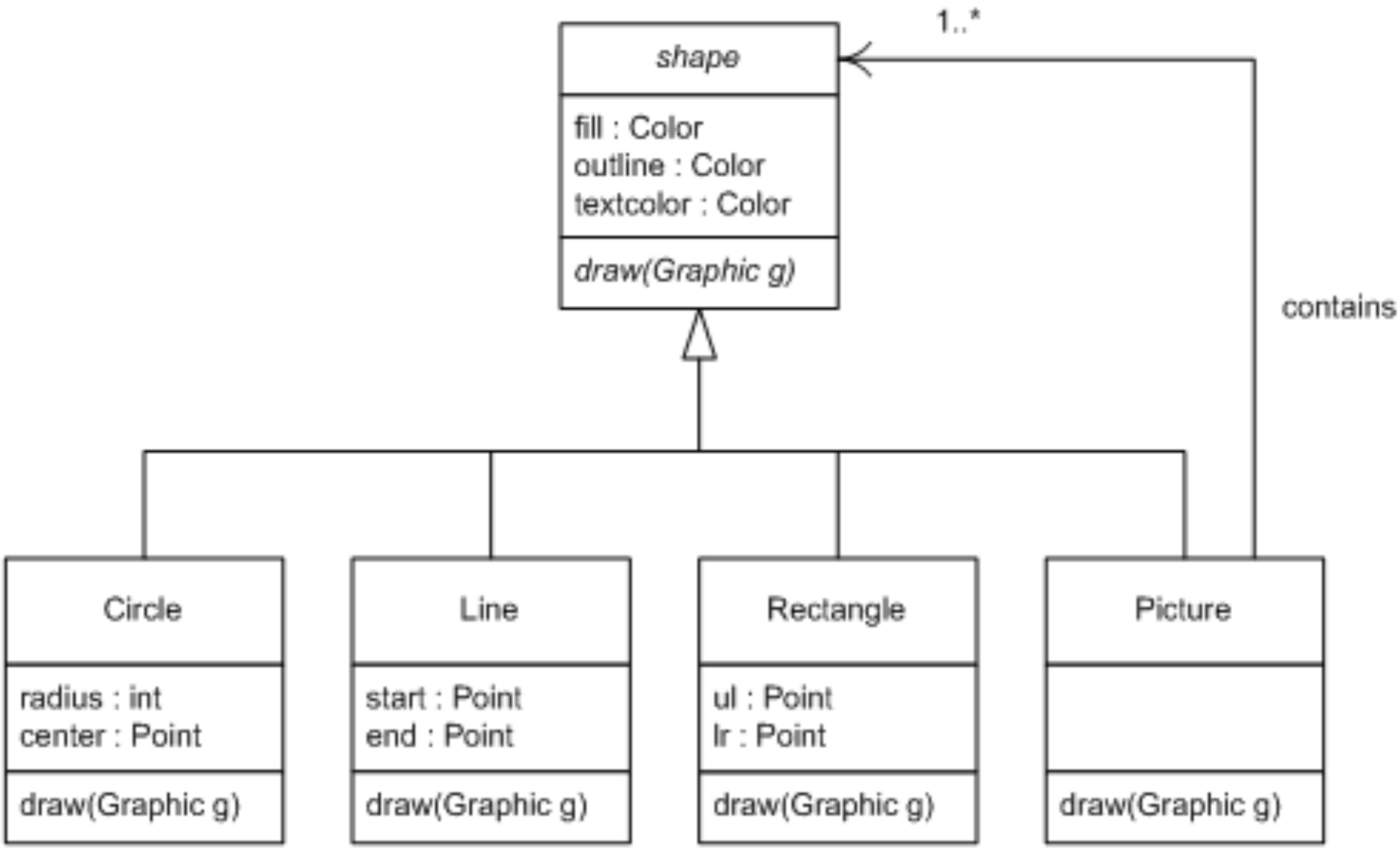
aComposite
aLeaf
aComposite
aLeaf
aLeaf
aLeaf
aLeaf
aLeaf

Composite



Client
Component
+ operation()
0..*
child
Leaf
+ operation()
Composite
- children : Components
+ operation()
+ add()
+ remove()
+ getChild()
1
parent
for each child
child.operation()

Composite



1..*
shape
fill : Color
outline : Color
textcolor : Color
draw(Graphic g)
contains
Circle
radius : int
center : Point
draw(Graphic g)
Line
start : Point
end : Point
draw(Graphic g)
Rectangle
ul : Point
lr : Point
draw(Graphic g)
Picture
draw(Graphic g)

Definition

Definition

"Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly."

-Gang of Four

Definition

- **Name** "Composite"
  - A recursive, whole-part hierarchy
  - In practice, a Tree structure

- **Intent**
  - Let a client treat a part of a composition in the same manner as they would treat the whole composition
  - Make the construction *transparent* with regards to how individual parts are treated
    - *Primitives have same interface as compositions*

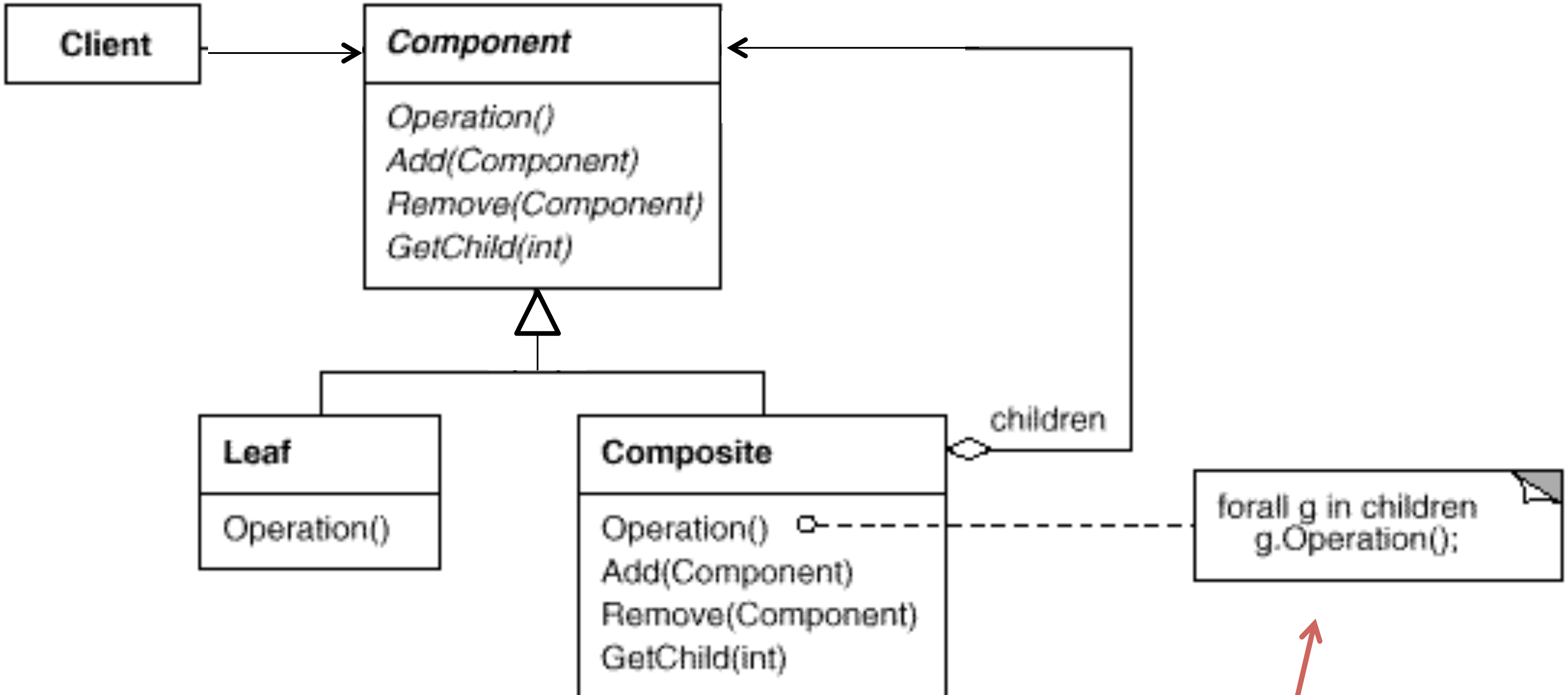
How

Composite Pattern - Participants

Participants

- **Component**
  - Declares the interface for objects in the composition
  - Implements default behavior common to all classes
  - Declares an interface for accessing and managing children
- **Leaf**
  - Represents leaf objects in the composition. Has no children.
- **Composite**
  - Defines behavior for components having children
  - Stores child components
  - Implements child-related operations in the Component interface
- **Client**
  - Manipulates objects in the composition through the (common) Component interface

Composite Pattern - Structure

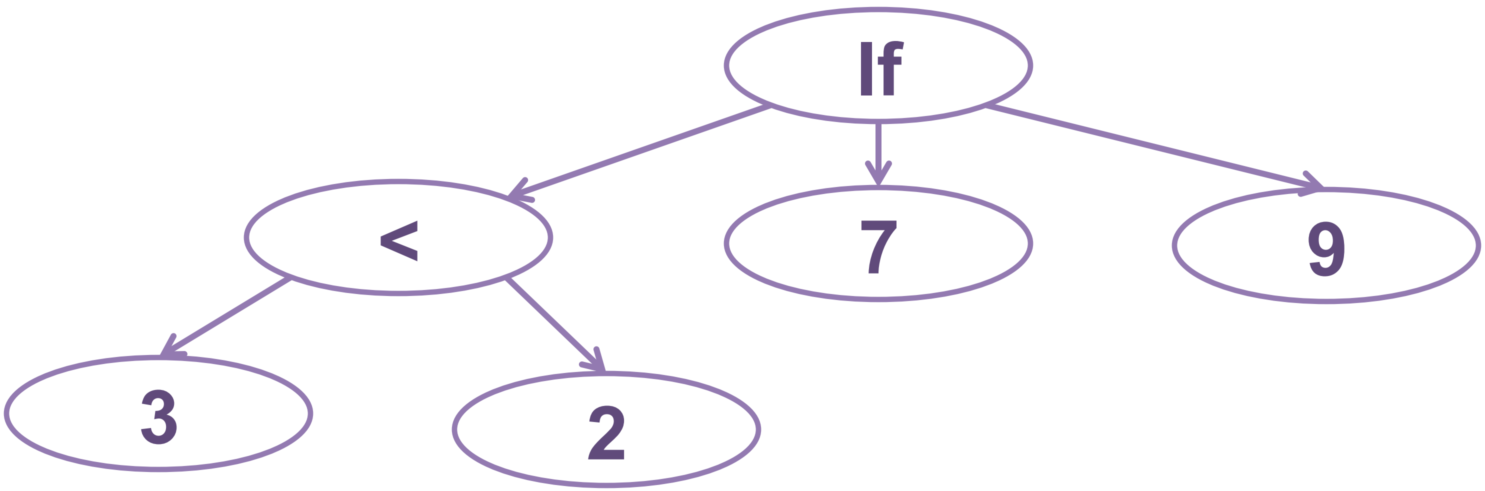


Note: Operation propagates to all children in a composite

Example
Program Interpreter

Example: Program Interpreter

- A bit of a review of 3155…
  - A program can be described in terms of an Abstract Syntax Tree
  - Every node in the tree is a sub-expression
  - Evaluating the program involves recursively evaluating the sub-expressions, until leaves are reached



Example: Program Interpreter

- We can model every node using some *Expression* object
  - Each node has an `evaluate()` method
  - `evaluate()` recursively calls `evaluate()` on the children of the node
  - Code is "executed" by evaluating the root node

- Treat every node as an *Expression* object
  - Literals (numbers, booleans, etc.) become ***Leaf*** objects
  - Operators (if, +, etc.) become ***Composite*** objects
  - *Expression* is the ***Component***
  - ***Client*** can call *evaluate()* on <u>*any*</u> *Expression*

Program Interface

Why treat things this way?

- Literals are already represented in our host language, so why wrap these in an object of some sort?
- Consider two implementations…
  - *Treat literals as literals*
  - *Treat literals as leafs*

Program Interface

– Treat literals as literals

```
class If: public Expression
{
    int evaluate(bool b1, Expression e2, Expression e3) { … }
    int evaluate(Expression e1, int n1, Expression e3) { … }
    …
    bool evaluate(bool b1, Expression e2, bool b3) { … }
    …
}
```

- *# of argument combinations (Expression, bool, int)*
  = 81 <u>unique method signatures</u>

Program Interface

– How to handle ambiguous return type?

```
int evaluate(bool b1, Expression e2, Expression e3)
bool evaluate(bool b1, Expression e2, Expression e3)
```

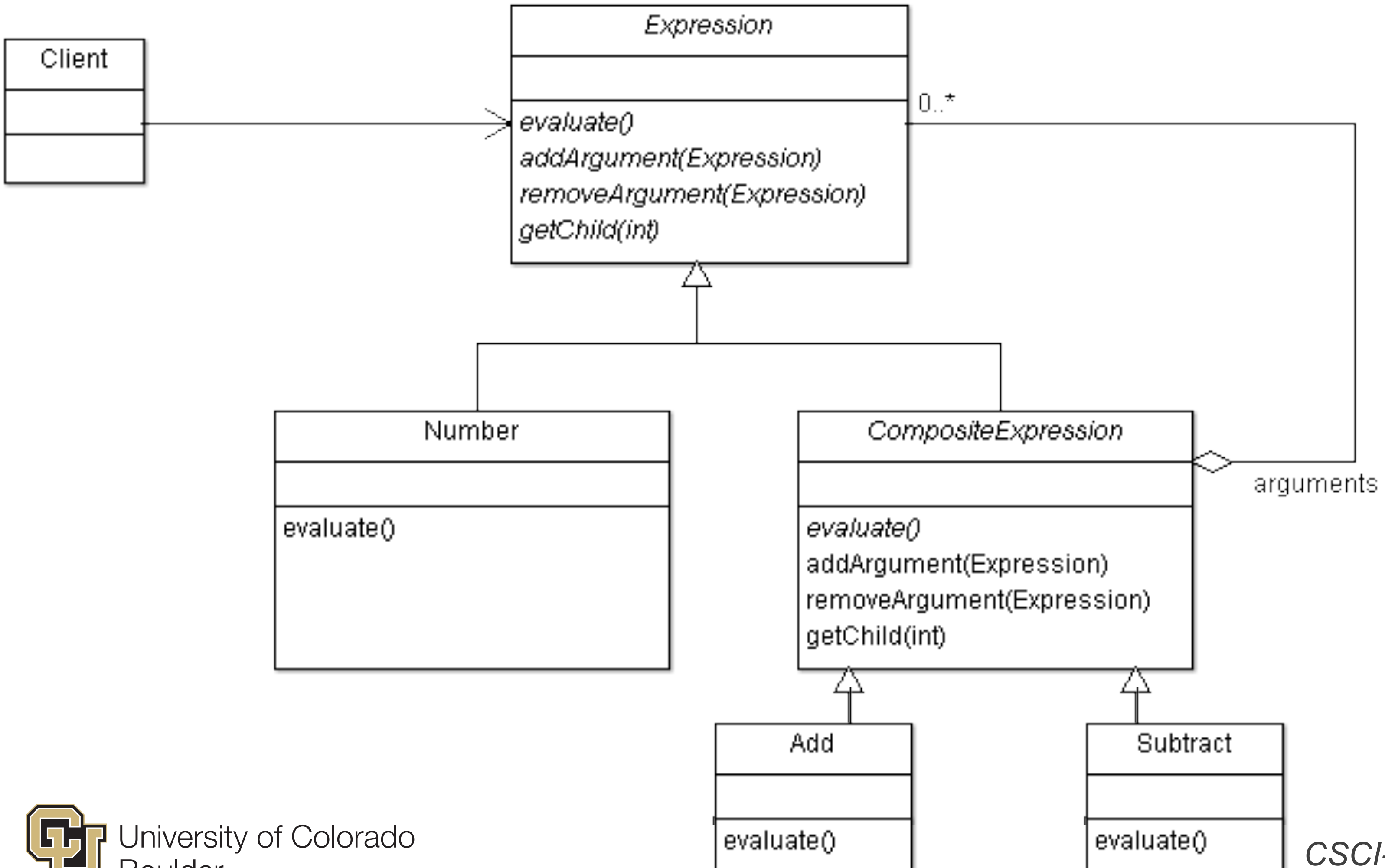
– Treat literals as Leafs

```
class If: public Expression
{
   Expression evaluate(Expression e1, Expression e2, Expression e3)
    { … }
}
```

- *Single signature for evaluate, since* all *components are treated as Expression objects (expressions and literals alike)*

Program Interface - Composite

Problem: Create a simple interpreter to evaluate arithmetic expressions (just add and multiply)



Design Considerations

Consequences

- Wherever client code expects a primitive object, it can also take a composite object
  - A Number (say, 10), can be treated as though it were an Expression (say, 5+5)
- Simplifies client code
  - All structures are treated the same, single method signature to interact with everything
- New components can be easily added
  - Just subclass Leaf!
- Makes design overly general
  - Difficult to restrict the components of a composite
  - Cannot rely on type system to enforce constraints, much use run-time checks instead

Implementation Considerations

- ***Component*** class maintains the *common* interface for ***Leaf*** and ***Composite***
  - The goal is to *maximize* this interface
  - Issues: This would introduce operations into *Leaf* classes which should not be there
    - *myLeaf->getChild(3) ??!?!*
  - Why did this come up?
    - *We're trading off* transparency *for* safety*.*
  - How should this be handled?
    - *Ignore the request?*
    - *Have a default response in Component?*
    - *Throw an exception!!!*

Implementation Considerations

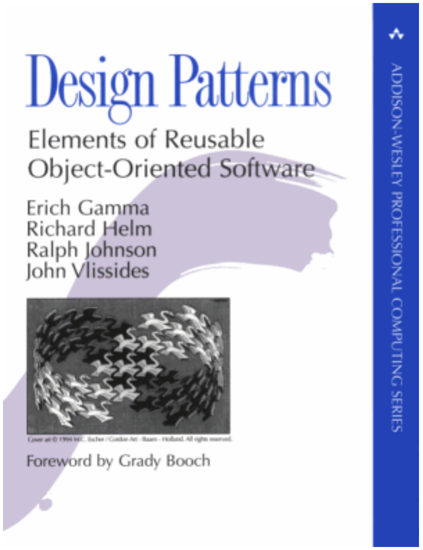
- Should Component implement a list of Components?
  - Why not simply combine Component and Composite classes?
    - *Leaf subclasses Component*
    - *If Component has a List structure as an attribute, so does* every *Leaf!*
  - How many Leaf objects in the Tree?
    - *Create / store a List for all those Leaf objects*
    - *Potentially BIG memory problem*
- Caching
  - The results of a common operation may be stored in one of the *Composite* instances
    - *No need to recalculate if this result doesn't change!*
    - *Short circuit the child->operation() call.*

Comparisons

- **Composite** and **Decorator** have similar structure diagrams, reflecting the fact that both rely on recursive composition to organize an open-ended number of objects.
- Can traverse a Composite using **Iterator**
- **Flyweight** frequently combined with Composite.
- **Decorator** is designed to let you add responsibilities to objects without subclassing. Composite's focus is not on embellishment but on representation. These intents are distinct but complementary. Consequently, Composite and Decorator are often used in concert.
- It could use **Observer** to tie one object structure to another and **State** to let a component change its behavior as its state changes.

https://sourcemaking.com/design_patterns/composite

Further Reading



- **Design Patterns**
  pp. 163 - 173