# Design Patterns

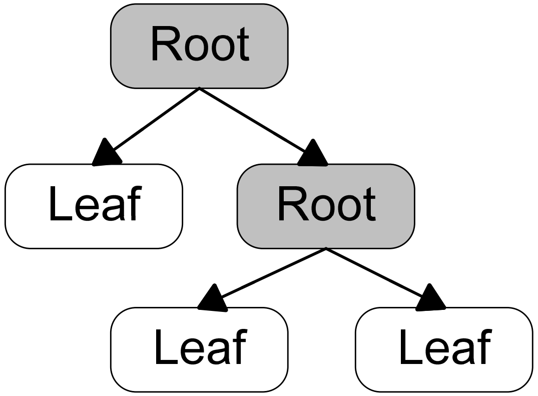
宋 杰
Song Jie
东北大学 软件学院
Software College, Northeastern University

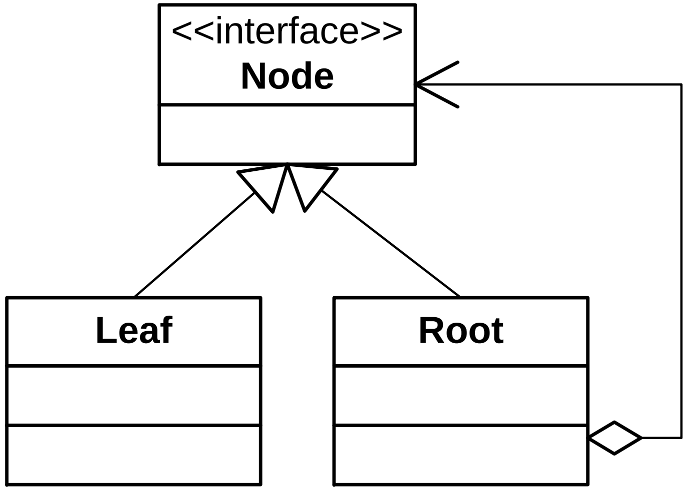
# 8. Composite Pattern

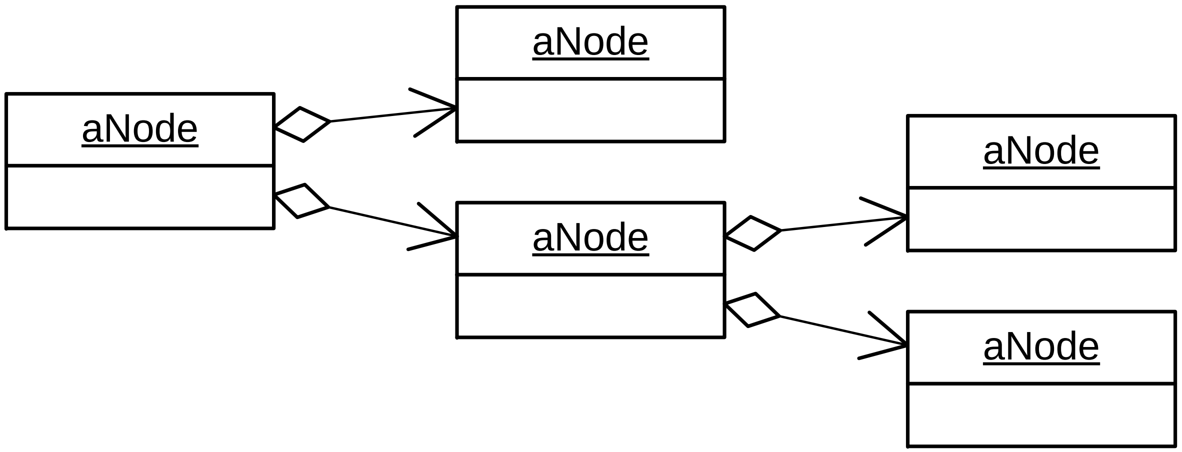
# Intent

- Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- 将对象组合成树形结构以表示“部分-整体”的层次结构。合成模式使得用户对单个对象(叶子节点，单纯元素)和组合对象(根节点，复合元素)的使用具有一致性。

# Example


Root
Leaf
Root
Leaf
Leaf


<<interface>>
Node
Leaf
Root


aNode
aNode
aNode
aNode
aNode

# Version 1-1

```java
interface Node {
    public void operation();
}

class Leaf implements Node {

    public void operation() {
        // do Leaf's operation
    }
}

class NodeClient {
    public Node createTree() {
        Root rootA = new Root();
        Root rootB = new Root();
        Leaf leafA = new Leaf();
        Leaf leafB = new Leaf();
        Leaf leafC = new Leaf();
        rootA.addSubNode(rootB);
        rootA.addSubNode(leafA);
        rootB.addSubNode(leafB);
        rootB.addSubNode(leafC);
        return rootA;
    }
}
```

```java
class Root implements Node {
    private List<Node> nodeList;
    public Root() {
        nodeList = new ArrayList<Node>();
    }
    public void operation() {
        // pre-operations here
        for (Node node : nodeList) {
            node.operation();
        }
        // post-operations here
    }
    public void addSubNode(Node node) {
        nodeList.add(node);
    }
    public Node removeSubNodeByIndex(int index) {
        return nodeList.remove(index);
    }
    public void clearSubNodes() {
        nodeList.clear();
    }
    public int getSubNodesSize() {
        return nodeList.size();
    }
    public Node getSubNodeByIndex(int index) {
        return nodeList.get(index);
    }
}
```

# Version 1-2

```java
class NodeClient {
    public Node createTree() {
        Root rootA = new Root();
        Root rootB = new Root();
        Leaf leafA = new Leaf();
        Leaf leafB = new Leaf();
        Leaf leafC = new Leaf();
        rootA.addSubNode(rootB)
             .addSubNode(leafA);
        rootB.addSubNode(leafB)
             .addSubNode(leafC);
        return rootA;
    }
}
```

```java
class Root implements Node, Iterable<Node> {
    private List<Node> nodeList;
    public Root() {
        nodeList = new ArrayList<Node>();
    }
    public void operation() {
        // pre-operations here
        for (Node node : this) {
            node.operation();
        }
        // post-operations here
    }
    public Iterator<Node> iterator() {
        return nodeList.iterator();
    }
    public Root addSubNode(Node node) {
        nodeList.add(node);
        return this;
    }
    public Node removeSubNodeByIndex(int index) {
        return nodeList.remove(index);
    }
    public void clearSubNodes() {
        nodeList.clear();
    }
}
```

# Version 2

```
interface Node extends Iterable<Node> {
    public void operation();
    public Node addSubNode(Node node);
    public Node removeSubNodeByIndex(int index);
    public void clearSubNodes();
}

class Leaf implements Node {
    public void operation() {
        // do Leaf's operation
    }
    public Iterator<Node> iterator() {
        return new ArrayList<Node>().iterator();
    }
    public Node addSubNode(Node node) {
        return null;
    }
    public void clearSubNodes() {
    }
    public Node removeSubNodeByIndex(int index) {
        return null;
    }
}
```

# Version 2

```java
class NodeClient {
    public Node createTree() {
        Node rootA = new Root();
        Node rootB = new Root();
        Node leafA = new Leaf();
        Node leafB = new Leaf();
        Node leafC = new Leaf();
        rootA.addSubNode(rootB)
             .addSubNode(leafA);
        rootB.addSubNode(leafB)
             .addSubNode(leafC);
        return rootA;
    }
}
```

```java
class Root implements Node {
    private List<Node> nodeList;
    public Root() {
        nodeList = new ArrayList<Node>();
    }
    public void operation() {
        // pre-operations here
        for (Node node : this) {
            node.operation();
        }
        // post-operations here
    }
    public Iterator<Node> iterator() {
        return nodeList.iterator();
    }
    public Node addSubNode(Node node) {
        nodeList.add(node);
        return this;
    }
    public Node removeSubNodeByIndex(int index) {
        return nodeList.remove(index);
    }
    public void clearSubNodes() {
        nodeList.clear();
    }
}
```

# Version 3

```java
abstract class Node implements Iterable<Node> {
    protected Node parentNode;
    public void addParentNode(Node parentNode) {
        this.parentNode = parentNode;
    }
    public Node getParentNode() {
        return parentNode;
    }
    public void removeParentNode() {
        parentNode = null;
    }
    public abstract Node addSubNode(Node node);
    public abstract Node removeSubNodeByIndex(int index);
    public abstract void clearSubNodes();
    public abstract Iterator<Node> iterator();
    public abstract void operation();
}
```

# Version 3

```java
class Leaf extends Node {
    public Node addSubNode(Node node) {
        return null;
    }
    public void clearSubNodes() {
    }
    public Node removeSubNodeByIndex(int index) {
        return null;
    }
    public Iterator<Node> iterator() {
        return new ArrayList<Node>().iterator();
    }
    public void operation() {
        // do Leaf's operation
    }
}
```

# Version 3

```
class Root extends Node {
    private List<Node> nodeList;
    public Root() {
        this.nodeList = new ArrayList<Node>();
    }
    public void operation() {
        // do Root's operation
    }
    public Iterator<Node> iterator() {
        return nodeList.iterator();
    }
    public Root addSubNode(Node node) {
        nodeList.add(node);
        return this;
    }
    public Node removeSubNodeByIndex(int index) {
        return nodeList.remove(index);
    }
    public void clearSubNodes() {
        nodeList.clear();
    }
}
```

# Version 3

```
class NodeClient {
    public Node createTree() {
        Node rootA = new Root();
        Node rootB = new Root();
        Node leafA = new Leaf();
        Node leafB = new Leaf();
        Node leafC = new Leaf();

        rootB.addParentNode(rootA);
        leafA.addParentNode(rootA);
        leafB.addParentNode(rootB);
        leafC.addParentNode(rootB);

        rootA.addSubNode(rootB).addSubNode(leafA);
        rootB.addSubNode(leafB).addSubNode(leafC);

        return rootA;
    }

    public void ModifyTree() {
        Node tree = createTree();
        Node leafA = tree.removeSubNodeByIndex(0);
        leafA.removeParentNode();
    }
}
```

# Version 4

```
abstract class Node implements Iterable<Node> {
    protected Node parentNode;
    protected void addParentNode(Node parentNode) {
        this.parentNode = parentNode;
    }
    protected void removeParentNode() {
        parentNode = null;
    }
    public Node getParentNode() {
        return parentNode;
    }

    public abstract Node addSubNode(Node node);
    public abstract Node removeSubNodeByIndex(int index);
    public abstract void clearSubNodes();
    public abstract Iterator<Node> iterator();
    public abstract void operation();
}
```

# Version 4

```
class Leaf extends Node {
    public Node addSubNode(Node node) {
        return null;
    }
    public void clearSubNodes() {
    }
    public Node removeSubNodeByIndex(int index) {
        return null;
    }
    public Iterator<Node> iterator() {
        return new ArrayList<Node>().iterator();
    }
    public void operation() {
        // do Leaf's operation
    }
}
```

# Version 4

```
class Root extends Node {
    private List<Node> nodeList;
    public Root() {
        this.nodeList = new ArrayList<Node>();
    }
    public Node addSubNode(Node node) {
        nodeList.add(node);
        node.addParentNode(this);
        return this;
    }
    public Node removeSubNodeByIndex(int index) {
        Node node = nodeList.remove(index);
        node.removeParentNode();
        return node;
    }
    public void clearSubNodes() {
        for (Node node : this) {
            node.removeParentNode();
        }
        nodeList.clear();
    }
    public void operation() {
        // do Root's operation
    }
    public Iterator<Node> iterator() {
        return nodeList.iterator();
    }
}
```

# Version 4

```
class NodeClient {
    public Node createTree() {
        Node rootA = new Root();
        Node rootB = new Root();
        Node leafA = new Leaf();
        Node leafB = new Leaf();
        Node leafC = new Leaf();

        rootA.addSubNode(rootB).addSubNode(leafA);
        rootB.addSubNode(leafB).addSubNode(leafC);

        return rootA;
    }

    public void ModifyTree() {
        Node tree = createTree();
        tree.removeSubNodeByIndex(0);
    }
}
```

# Version 5

```java
interface Node extends Iterable<Node> {

    public boolean isLeaf();

    public void addParentNode(Node parentNode);
    public void removeParentNode();

    public Node getParentNode();
    public Node addSubNode(Node node);
    public Node removeSubNodeByIndex(int index);
    public void clearSubNodes();

    public Iterator<Node> iterator();
    public void operation();
}
```

# Version 5

```
class NodeImpl implements Node {
    private List<Node> nodeList;
    private Node parentNode;
    private boolean leaf;

    public NodeImpl() {
        this.nodeList = new ArrayList<Node>();
    }
    public NodeImpl(boolean isLeaf) {
        this();
        this.leaf = isLeaf;
    }

    public boolean isLeaf() {
        return leaf;
    }
    public void operation() {
        if (leaf) {
            // do Leaf's operation
        } else {
            // do Root's operation
        }
    }
    public Iterator<Node> iterator() {
        return nodeList.iterator();      }
    }
```

```
public void addParentNode(Node parentNode) {
    this.parentNode = parentNode;
}
public void removeParentNode() {
    parentNode = null;
}
public Node getParentNode() {
    return parentNode;
}
public Node addSubNode(Node node) {
    if(leaf) {return null;}
    nodeList.add(node);
    node.addParentNode(this);
    return this;
}
public Node removeSubNodeByIndex(int index) {
    if(leaf) {return null;}
    Node node = nodeList.remove(index);
    node.removeParentNode();
    return node;
}
public void clearSubNodes() {
    if(leaf) {return;}
    for (Node node : this) {
        node.removeParentNode();
    }
    nodeList.clear();
}
```

# Version 5

```
class NodeClient {
    public Node createTree() {
        Node rootA = new NodeImpl(false);
        Node rootB = new NodeImpl(false);
        Node leafA = new NodeImpl(true);
        Node leafB = new NodeImpl(true);
        Node leafC = new NodeImpl(true);

        rootA.addSubNode(rootB)
             .addSubNode(leafA);
        rootB.addSubNode(leafB)
             .addSubNode(leafC);

        return rootA;
    }

    public void ModifyTree() {
        Node tree = createTree();
        tree.removeSubNodeByIndex(0);
    }
}
```

# Version 6

```
interface Node extends Iterable<Node> {
    public boolean isRoot();
    public void addParentNode(Node parentNode);
    public void removeParentNode();
    public Node getParentNode();
    public Node addSubNode(Node node);
    public Node removeSubNodeByIndex(int index);
    public void clearSubNodes();
    public Iterator<Node> iterator();
    public void operation();
}
```

# Version 6

```java
class NodeImpl implements Node {
    private List<Node> nodeList;
    private Node parentNode;
    private boolean root;
    public NodeImpl() {
        this.nodeList = new ArrayList<Node>();
    }
    public boolean isRoot() {
        return root;
    }
    public void operation() {
        if (root) {
            // do Root's operation
        } else {
            // do Root's operation
        }
    }
    public Iterator<Node> iterator() {
        return nodeList.iterator();
    }
    public void addParentNode(Node parentNode) {
        this.parentNode = parentNode;
    }
    public void removeParentNode() {
        parentNode = null;
    }
    public Node getParentNode() {
        return parentNode;
    }
    public Node addSubNode(Node node) {
        root = true;
        nodeList.add(node);
        node.addParentNode(this);
        return this;
    }
    public Node removeSubNodeByIndex(int index) {
        Node node = null;
        if (root) {
            node = nodeList.remove(index);
            node.removeParentNode();
        }
        root = !nodeList.isEmpty();
        return node;
    }
    public void clearSubNodes() {
        if (root) {
            for (Node node : this) {
                node.removeParentNode();
            }
        }
        nodeList.clear();
        root = false;
    }
}
```

# Version 6

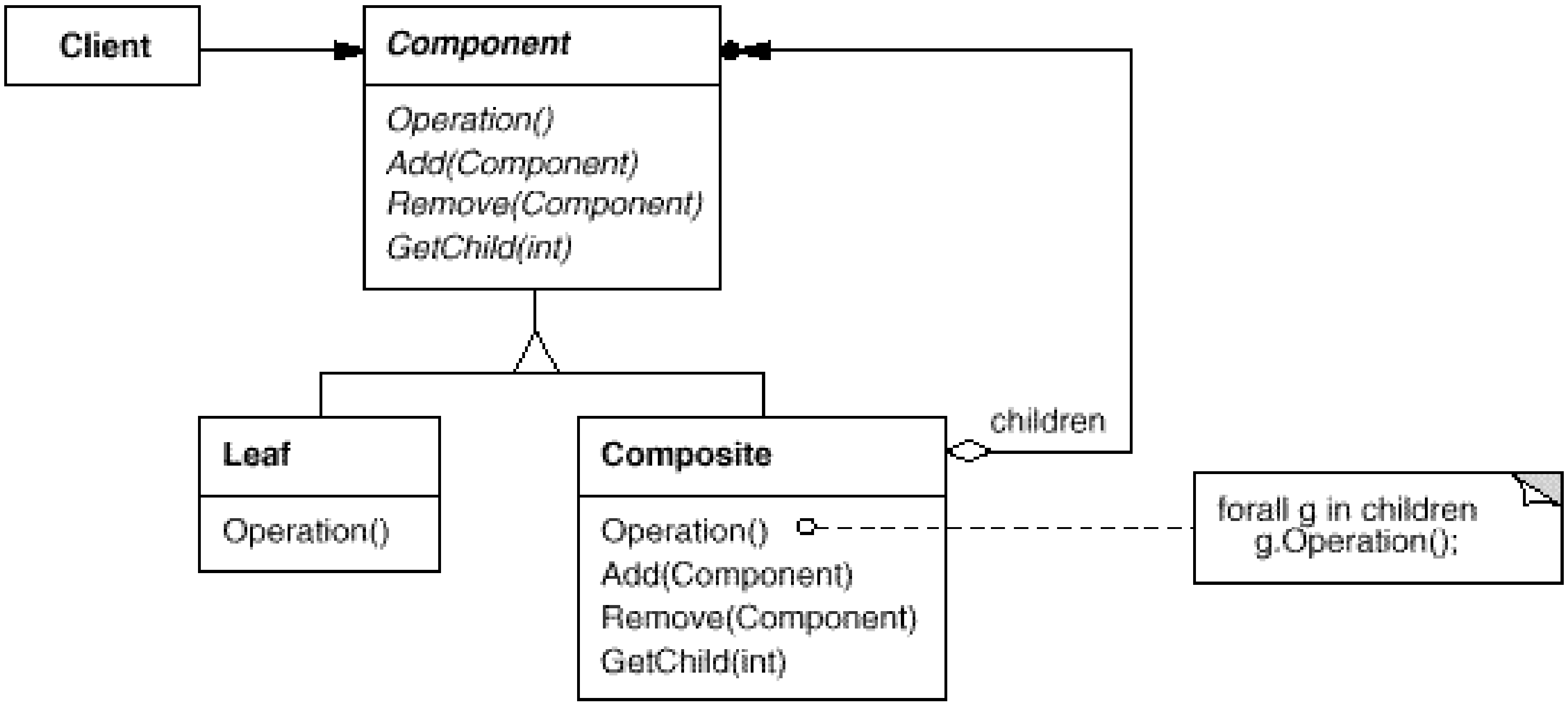
```
class NodeClient {
    public Node createTree() {
        Node rootA = new NodeImpl();
        Node rootB = new NodeImpl();
        Node leafA = new NodeImpl();
        Node leafB = new NodeImpl();
        Node leafC = new NodeImpl();

        rootA.addSubNode(rootB).addSubNode(leafA);
        rootB.addSubNode(leafB).addSubNode(leafC);

        return rootA;
    }

    public void ModifyTree() {
        Node tree = createTree();
        tree.removeSubNodeByIndex(0);
    }
}
```

# Structure



Client
Component
Operation()
Add(Component)
Remove(Component)
GetChild(int)
children
Leaf
Operation()
Composite
Operation()
Add(Component)
Remove(Component)
GetChild(int)
forall g in children
g.Operation();

# Participants

- Component
  - Declares the interface for objects in the composition.
  - Implements default behavior for the interface common to all classes, as appropriate.
  - Declares an interface for accessing and managing its child components.
  - (*Optional*) Defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate.
- Leaf
  - Represents leaf objects in the composition. A leaf has no children.
  - Defines behavior for primitive objects in the composition.
- 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 Component interface.

# Collaborations

- Clients use the Component to interact with objects in the composite structure.
  - If the recipient is a Leaf, then the request is handled directly.
  - If the recipient is a Composite, then it usually forwards requests to its child components, possibly performing additional operations before and/or after forwarding.

# Consequences

- Defines class hierarchies consisting of primitive objects and composite objects.
- Makes the client simple.
  - Clients can treat composite structures and individual objects uniformly.
  - Clients normally don't know (and shouldn't care) whether they're dealing with a leaf or a composite component.
- Makes it easier to add new kinds of components.
  - Newly defined Composite or Leaf subclasses work automatically with existing structures and client code.
  - Clients don't have to be changed for new Component classes.

# Consequences

- Can make your design overly general.
  - The disadvantage that it is harder to restrict the components of a composite.
- Sometimes you want a composite to have only certain components. With Composite, you can't rely on the type system to enforce those constraints for you. You'll have to use run-time checks instead.

# Applicability

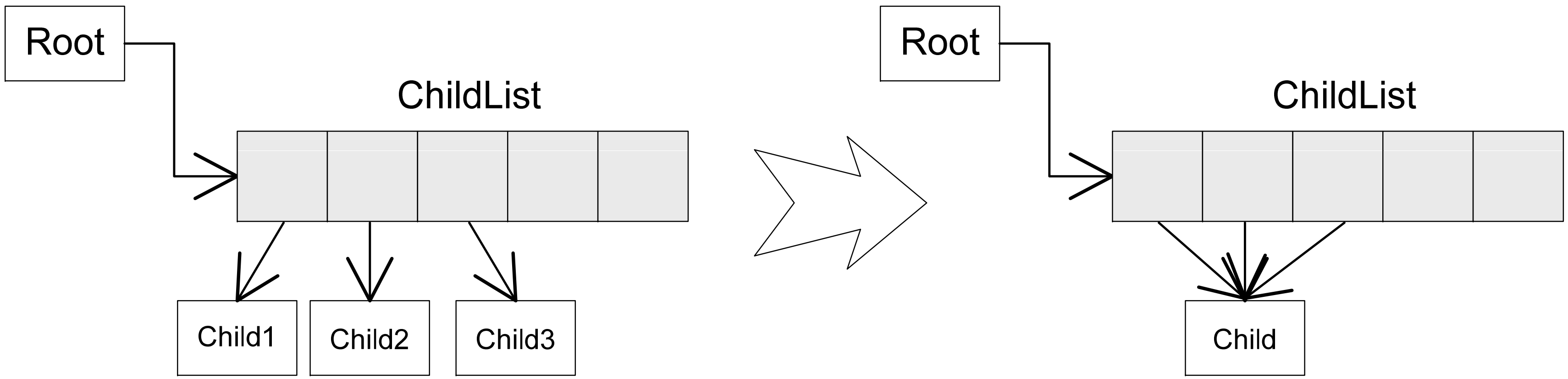
- You want to represent part-whole hierarchies of objects.
- You want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.

# Implementation 1: Explicit parent references (bi-direction reference )

- Maintaining references from child components to their parent can simplify the traversal and management of a composite structure.
  - The usual place to define the parent reference is in the Component class. Leaf and Composite classes can inherit the reference and the operations that manage it.
- It is unnecessary to let clients maintain bi-directions. Usually parent-to-children references are maintained by clients, child-to-parent reference are maintained inside composite pattern automatically
  - The easiest way to ensure this is to change a component's parent only when it's being added or removed from a composite. If this can be implemented once in the *Add* and *Remove* operations of the Composite class.

# Implementation 2: Sharing components

- It's often useful to share components, for example, to reduce storage requirements.
  - The component must be stateless or sharable state.



# Implementation 3: Maximizing the Component interface

- Composite pattern makes clients unaware of the specific Leaf or Composite classes they're using.
  - Component class should define as many common operations for Composite and Leaf classes as possible.
  - There are many operations that Component supports that don't seem to make sense for Leaf classes. So that it conflict with Interface Segregation Principle (ISP) ,also with Liskov Substitution Principle (LSP).
- How can Leaf provide a default implementation for them?
  - Make the useless operations, do nothing, or return null, or return mock object, or throws exception
- The child management operations are more troublesome and are discussed in the next item.

# Implementation 4: Declaring the child management operations

- Should we declare "*child management operations*" in the Component and make them meaningful for Leaf classes, or should we declare and define them only in Composite and its subclasses?
- The decision involves a trade-off between safety and transparency:
  - Transparency: Defining the child management interface at the root of the class hierarchy gives you transparency, because you can treat all components uniformly. It costs you safety, however, because clients may try to do meaningless things like add and remove objects from leaves.
  - Safety : Defining child management in the Composite class gives you safety, because any attempt to add or remove objects from leaves will be caught at compile-time in a statically typed language. But you lose transparency, because leaves and composites have different interfaces.

# Implementation 5: Heavy component or light component operations

- Transparency solution
  - The heavy Component is suggested because it can stand for the operations of both Leaf and Composite
- Safety solution
  - The light Component is suggested to let it only stand for the common operations of both Leaf and Composite

# Implementation 6: Child ordering

- Many designs specify an ordering on the children of Composite.
- When child ordering is an issue, you must design child access and management interfaces carefully to manage the sequence of children.

# Implementation 7: Caching to improve performance

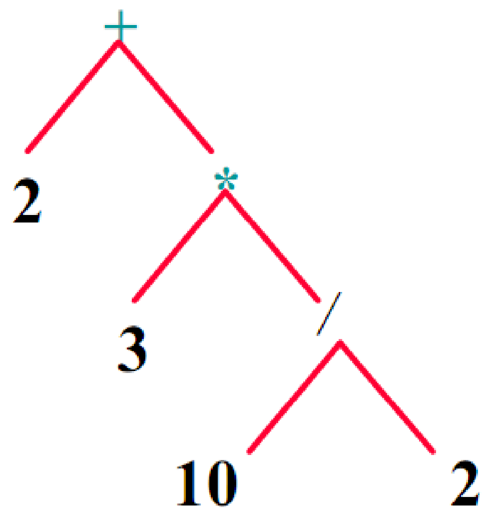
- If you need to traverse or search compositions frequently, the Composite class can cache traversal or search information about its children.
- Changes to a component will require invalidating the caches of its parents.
  - This works best when components know their parents.
- So if you're using caching, you need to define an interface for telling composites that their caches are invalid.

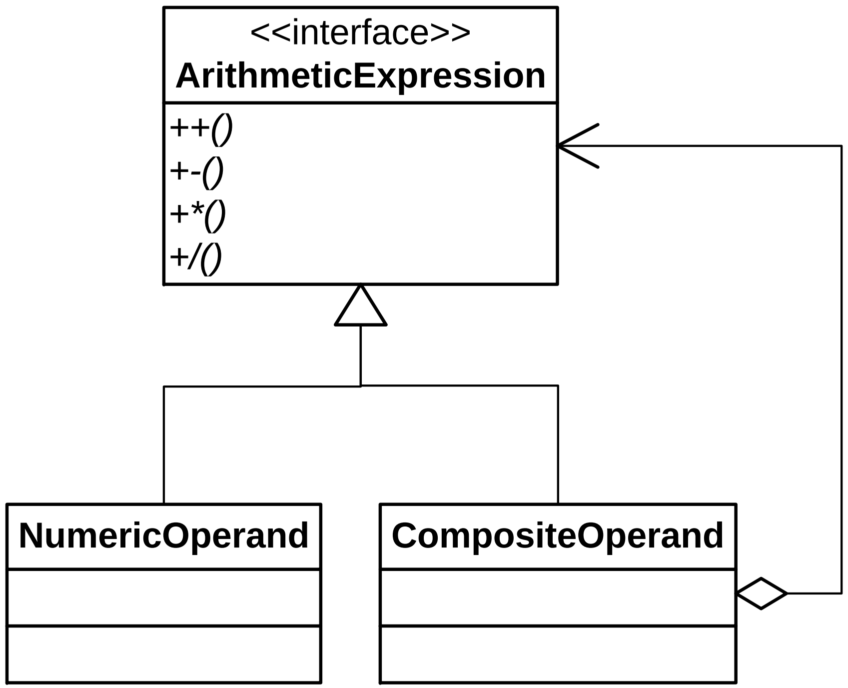
# Implementation 8: What's the best data structure for storing components?

- Composites may use a variety of data structures to store their children,
  - Arrays, List, Set, HashMap
- The choice of data structure depends (as always) on efficiency.
- Sometimes composites have a variable for each child (limited quantity).
  - binary tree: left and right

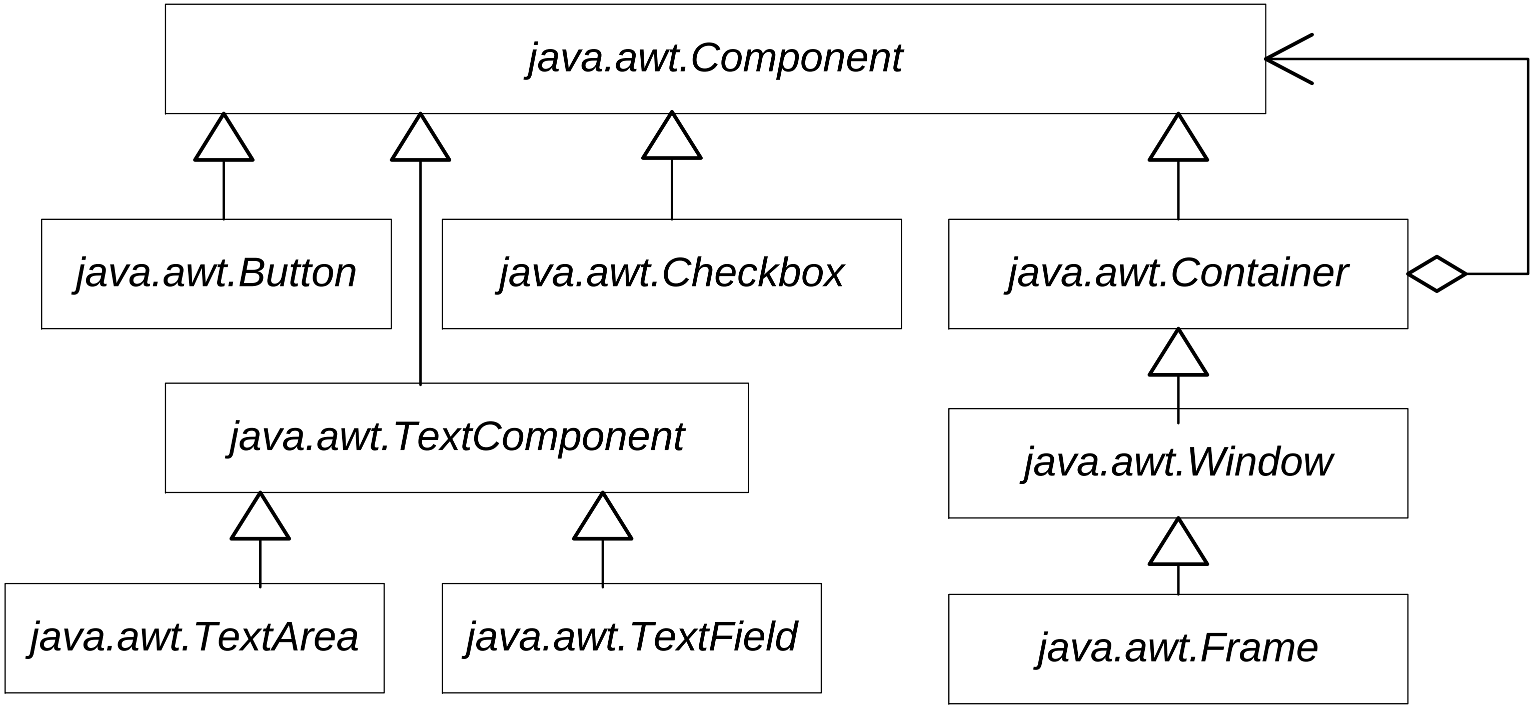
# Example 1: Arithmetic expressions

- Arithmetic expressions can be expressed as trees where an operand can be a number or an arithmetic expression.

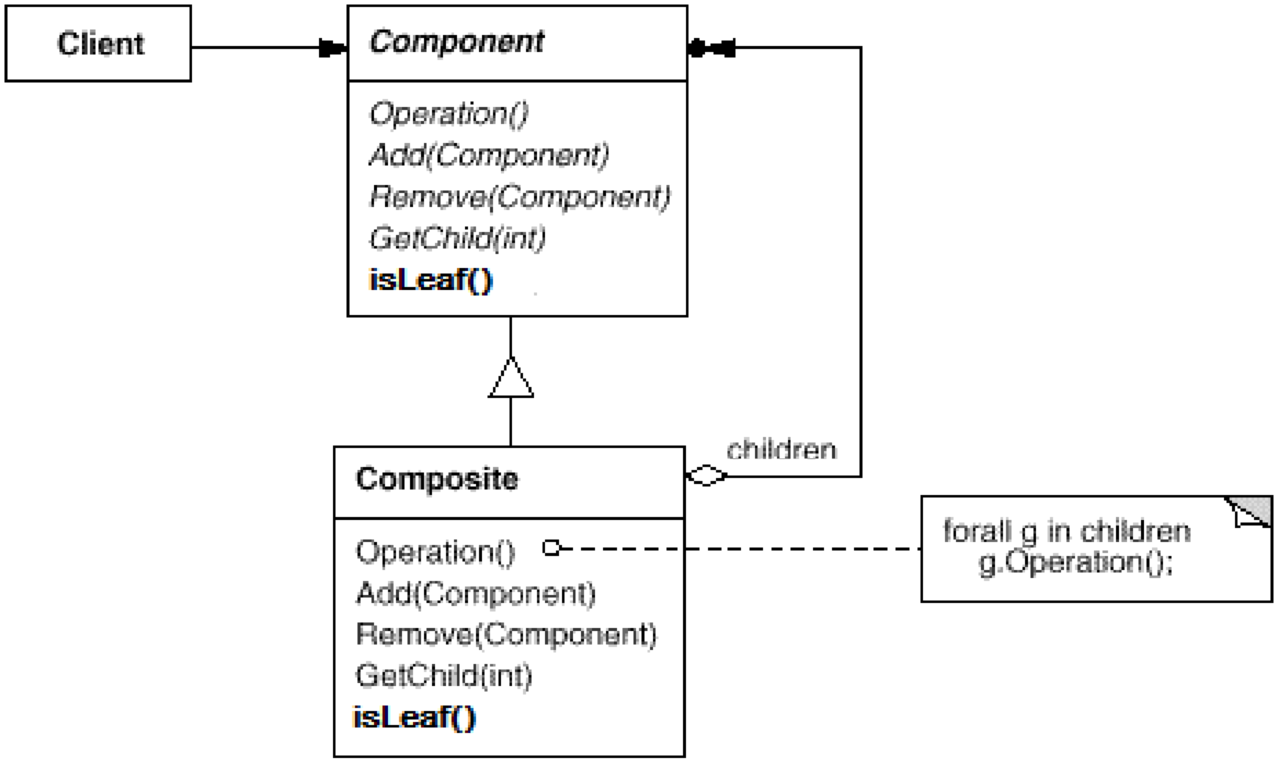


# Example 2: Java AWT

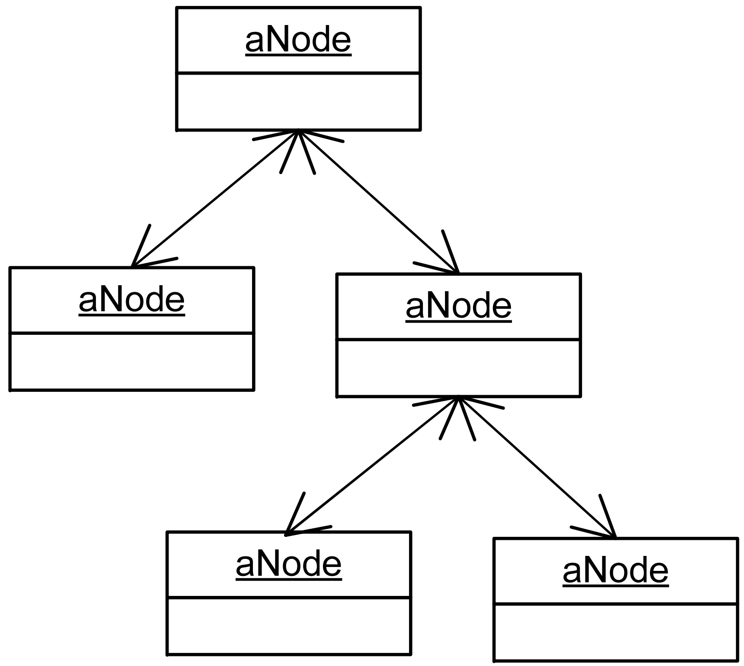


# Variation: Leaf and composite be one class



# Extension: Directions of the tree structure

- Top-down tree
- Bottom-up tree
- Bi-directions tree



Let’s go to next…