

# **Design Patterns and Language Evolution**

Ran Ettinger

Design Patterns, Software Engineering Programme

4 July 2003

**Claim: design patterns  
sometimes provide  
techniques to overcome  
language limitations**

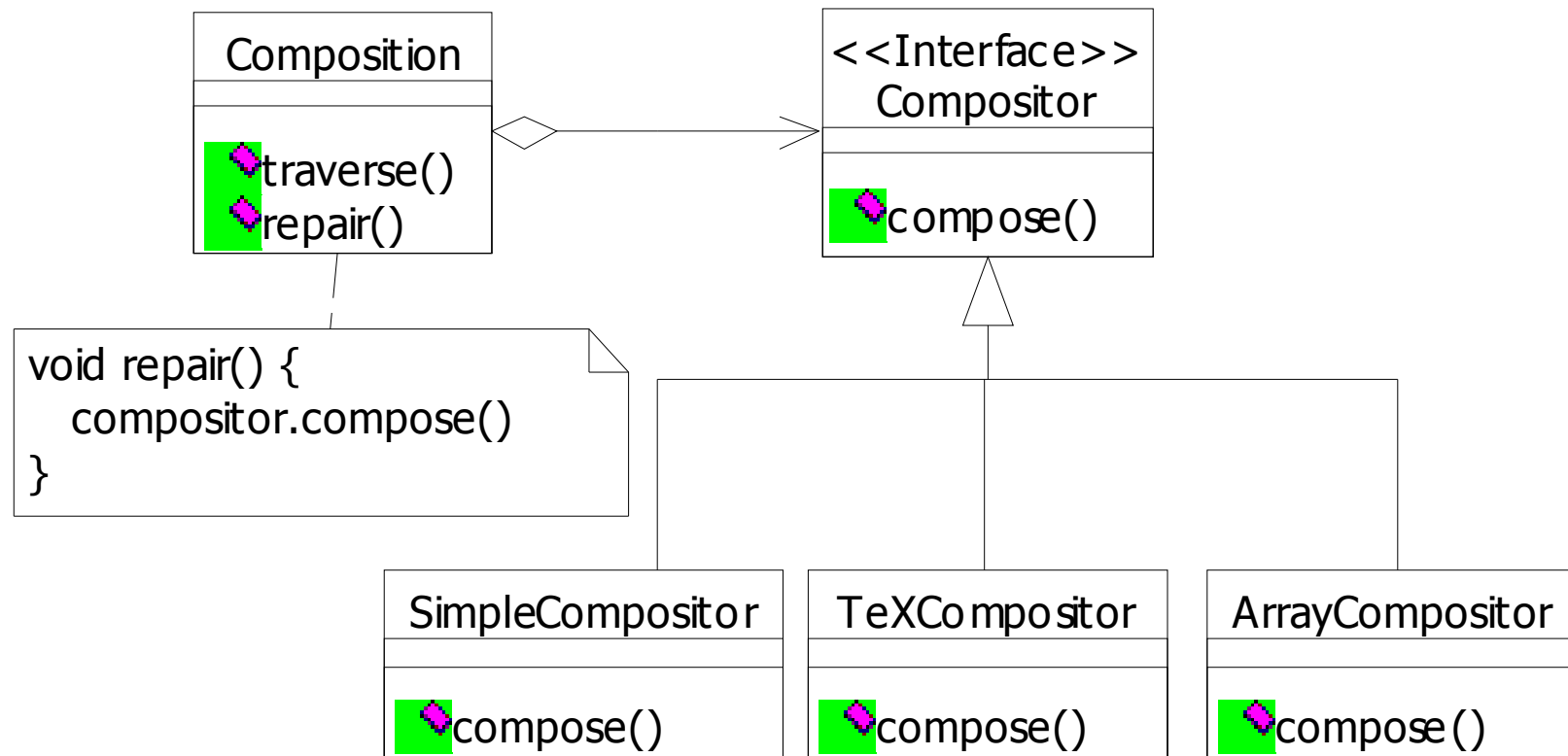
# Cases of Language Evolution

- ★ Polymorphism by virtual functions
- ★ AO method overriding mechanism
  - Both related to the **Strategy** pattern

# **Case 1: Polymorphism by virtual functions**

# Polymorphism

## GoF's text line breaking Strategy example

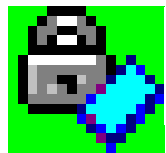


# Polymorphism (2)

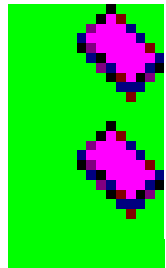
```
public static void main(String[] args) {  
    Composition quick = new Composition(new SimpleComposer());  
    Composition slick = new Composition(new TeXComposer());  
    Composition iconic = new Composition(new ArrayComposer());  
}
```

# No Polymorphism

Composition



compositionType



repair()

Composition()

# No Polymorphism (2)

```
public class Composition {  
    public static final int  
        SIMPLE=0, TEX=1, ARRAY=2;  
    private int compositorType;  
    public void repair() {  
        switch (compositorType) {  
            case SIMPLE:  
                // ...  
                break;  
            case TEX:  
                // ...  
                break;  
            case ARRAY:  
                // ...  
                break;  
        }  
    }  
}
```

```
public static void main(String[] args) {  
    Composition quick =  
        new Composition(SIMPLE);  
    Composition slick =  
        new Composition(TEX);  
    Composition iconic =  
        new Composition(ARRAY);  
}
```

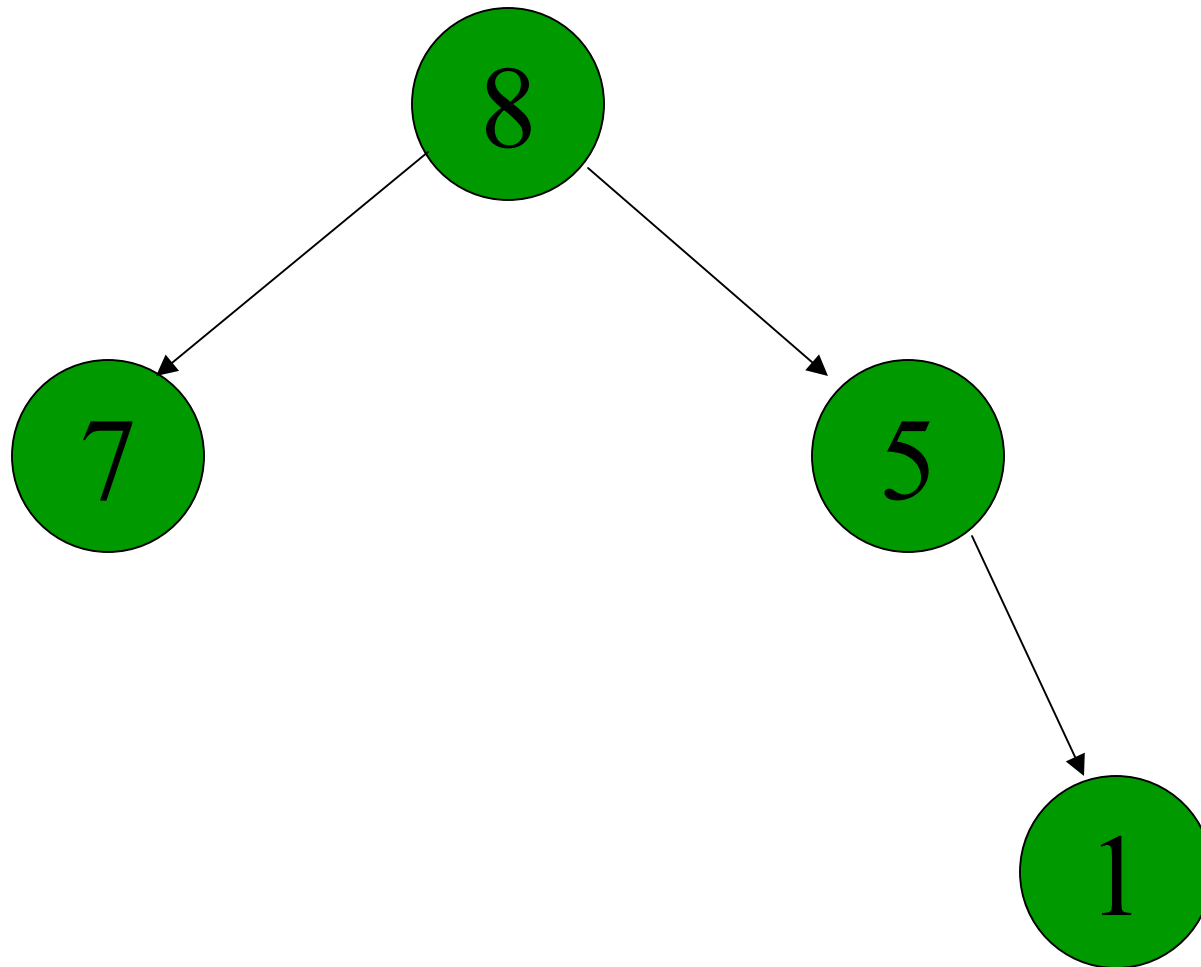
# **Case 2: AO method overriding mechanism**

# ***Software Configurability***

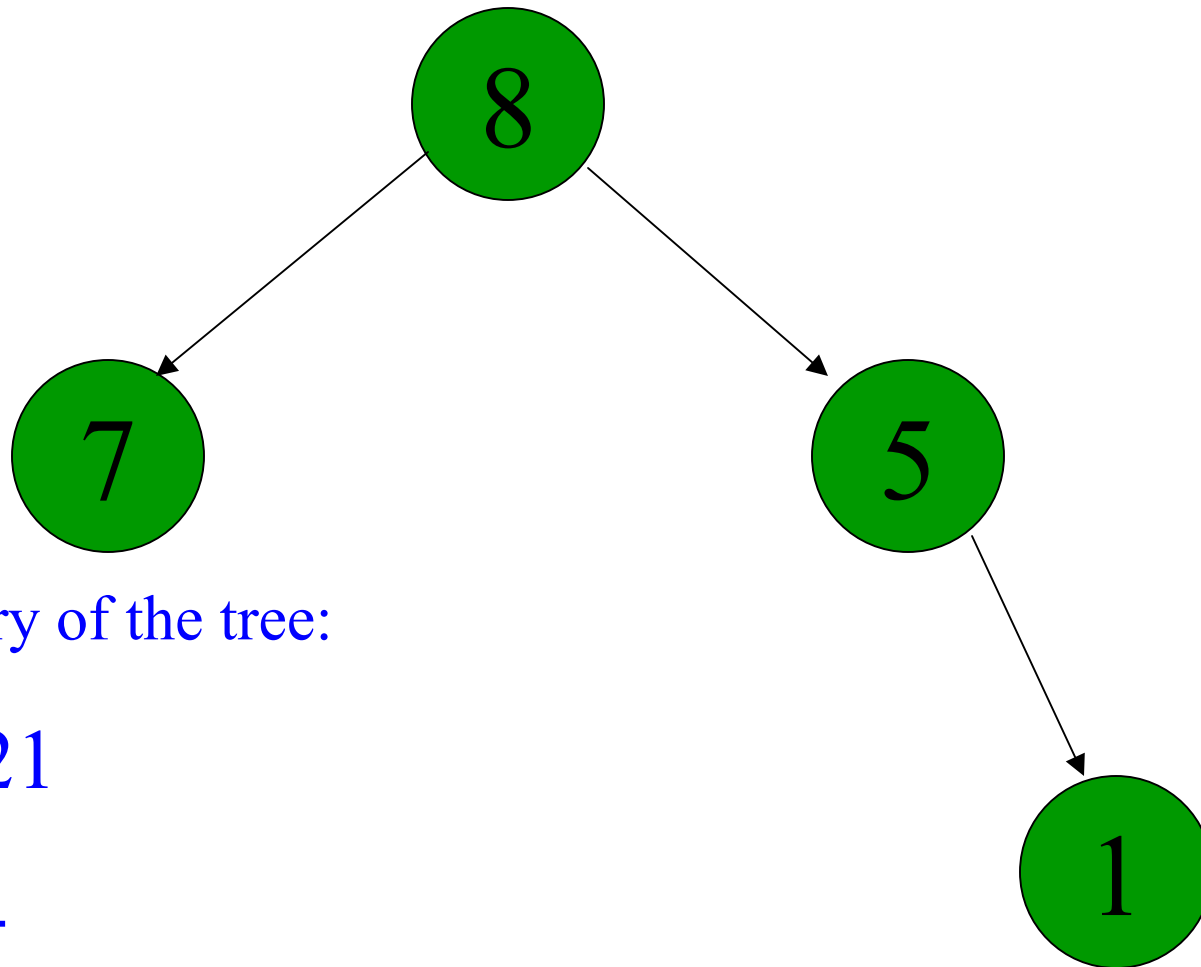
The ability to build, release and evolve a software system in a number of different configurations.

**How can high  
*configurability*  
improve software  
quality?**

# Example – Summary of Integer Binary Trees



# Standard Edition

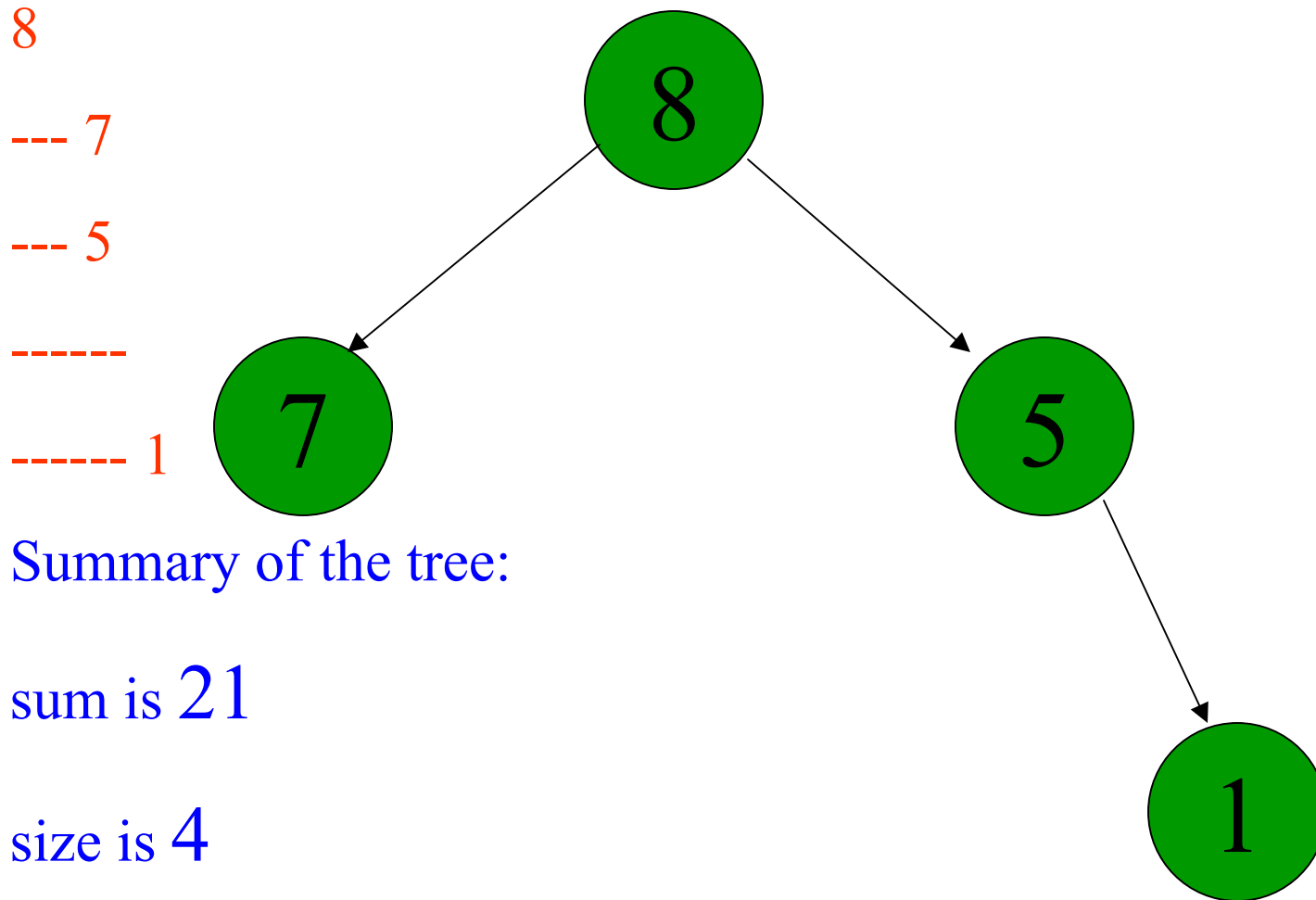


Summary of the tree:

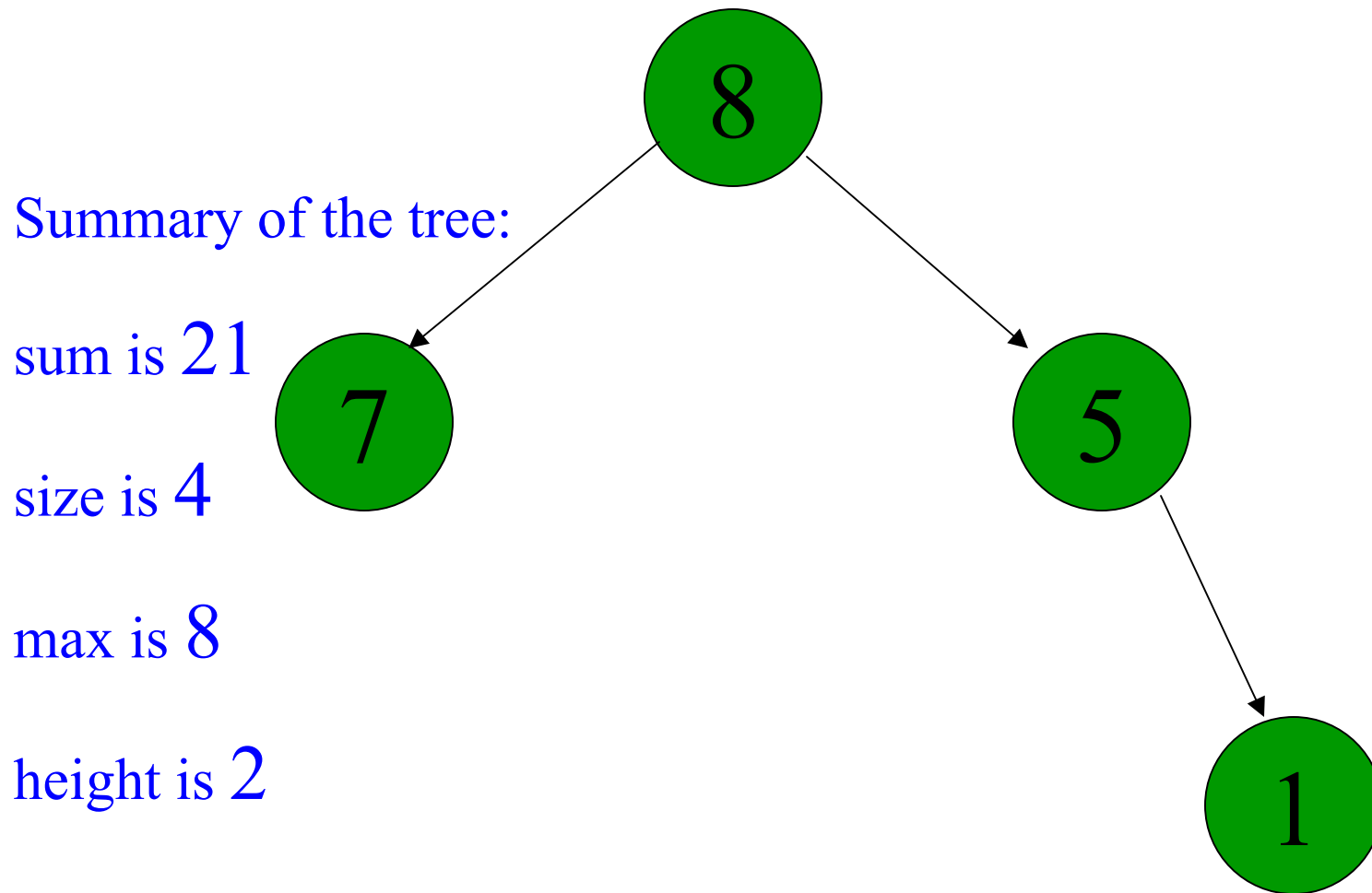
sum is 21

size is 4

# Standard <Debug> Edition



# Professional Edition



```
void computeSummary(Node node) {  
    int leftHeight=0;  
    height = 0;  
    if (node.left != null) {  
        computeSummary(node.left);  
        leftHeight = ++height;  
    }  
    if (node.right != null) {  
        computeSummary(node.right);  
        if (leftHeight > ++height)  
            height = leftHeight;  
    }  
    size++;  
    if (node.val > max)  
        max = val;  
    sum += val;  
}
```

**What are the OO  
*configurability*  
mechanisms and  
techniques ?**

# Pre-compiler directives

```
void computeSummary(Summary summary) {  
#ifdef HEIGHT  
    int leftHeight=0;  
    height = 0;  
#endif  
    if (node.left != null) {  
        computeSummary(node.left);  
#ifdef HEIGHT  
        leftHeight = ++height;  
#endif  
    }  
    if (node.right != null) {  
        computeSummary(node.right);  
#ifdef HEIGHT  
        if (leftHeight > ++height)  
            height = leftHeight;  
#endif  
    } ...  
}
```

# Pre-compiler directives

```
void computeSummary(Summary summary) {
```

```
#ifdef HEIGHT
```

```
    int leftHeight=0;
```

```
    height = 0;
```

```
#endif
```

```
    if (node.left != null) {
```

```
        computeSummary(node.left);
```

```
#ifdef HEIGHT
```

```
        leftHeight = ++height;
```

```
#endif
```

```
    }
```

```
    if (node.right != null) {
```

```
        computeSummary(node.right);
```

```
#ifdef HEIGHT
```

```
        if (leftHeight > ++height)
```

```
            height = leftHeight;
```

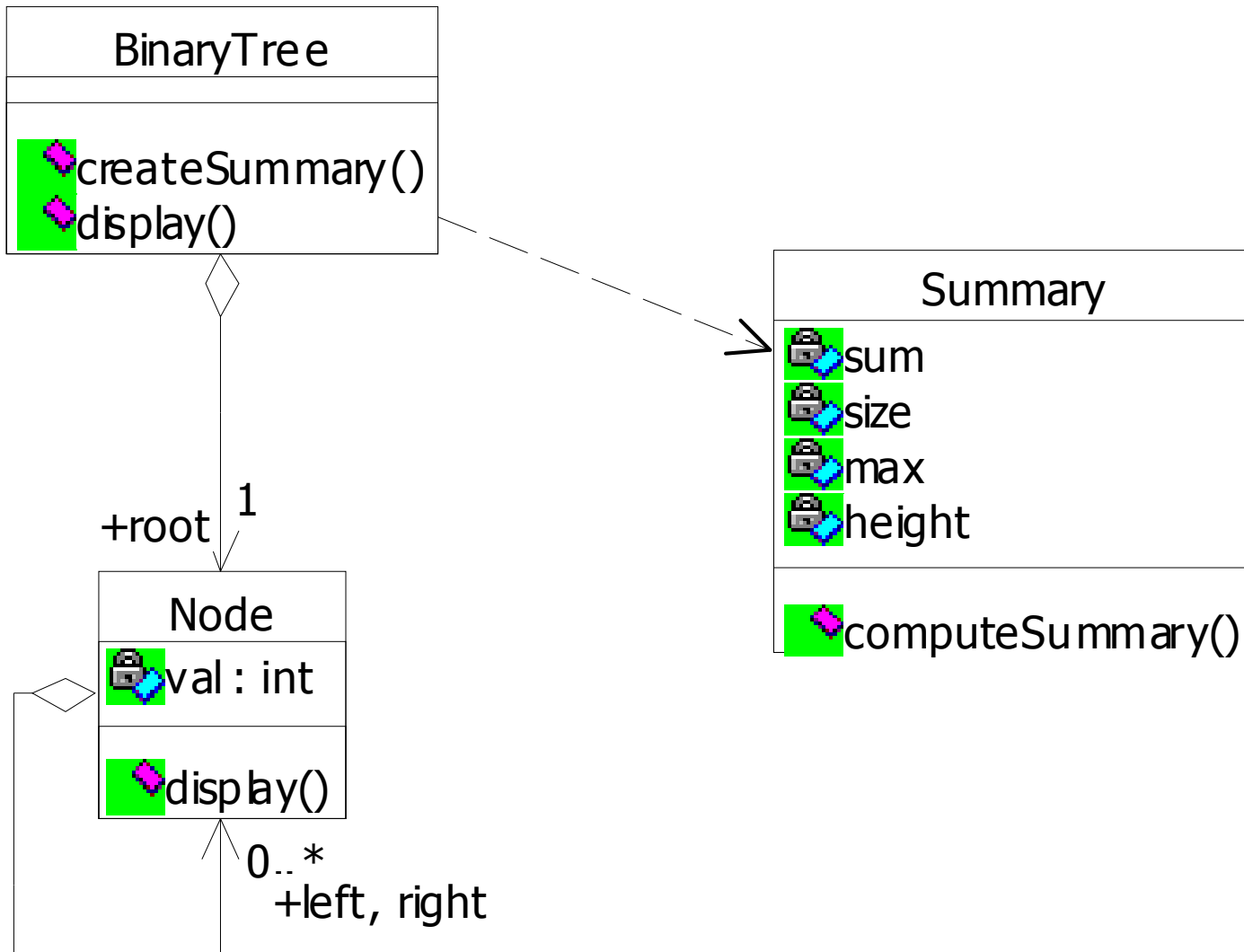
```
#endif
```

```
    } ...
```

```
}
```

```
#define HEIGHT
```

# Subclassing

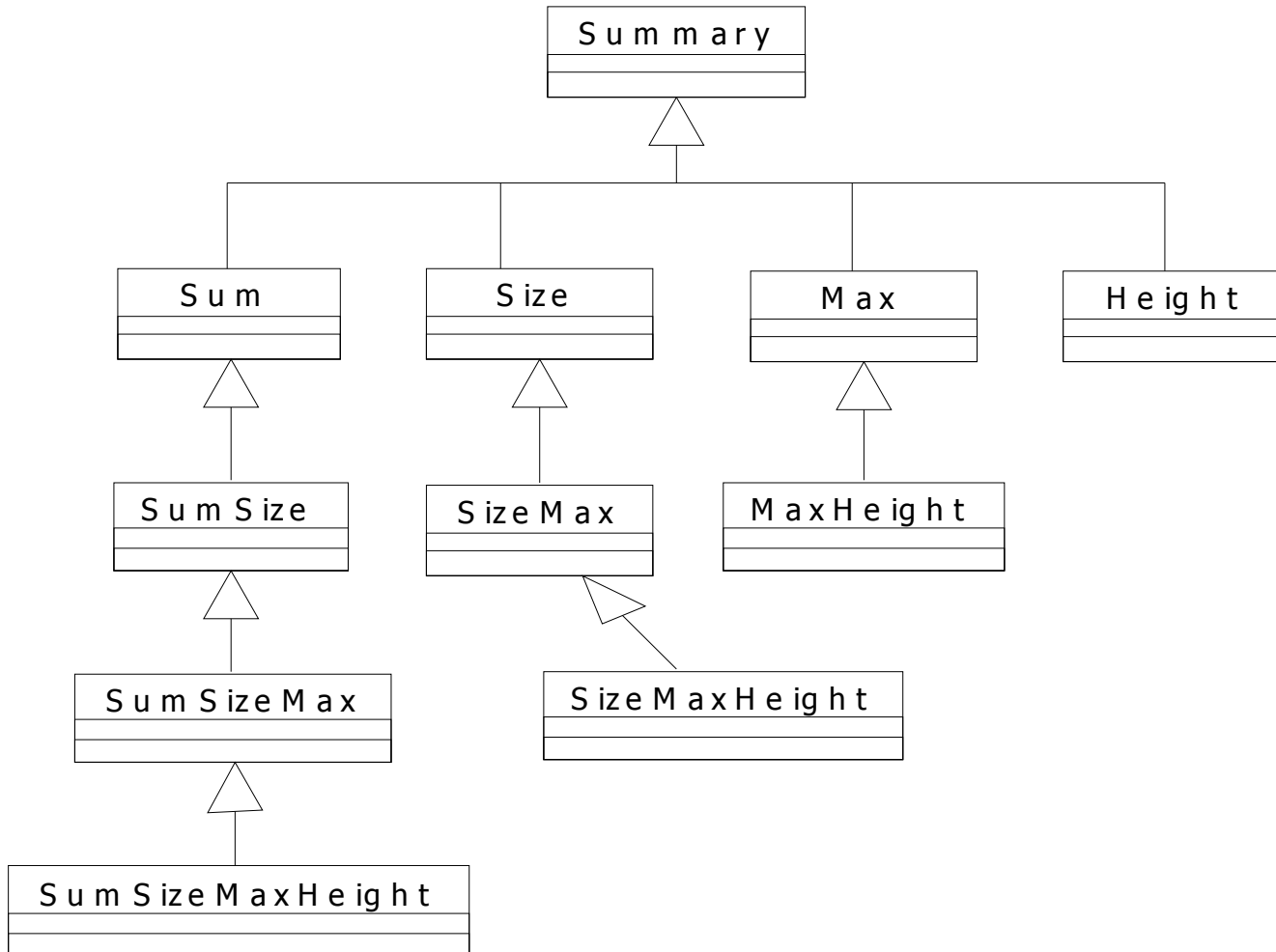


# Subclassing (2)

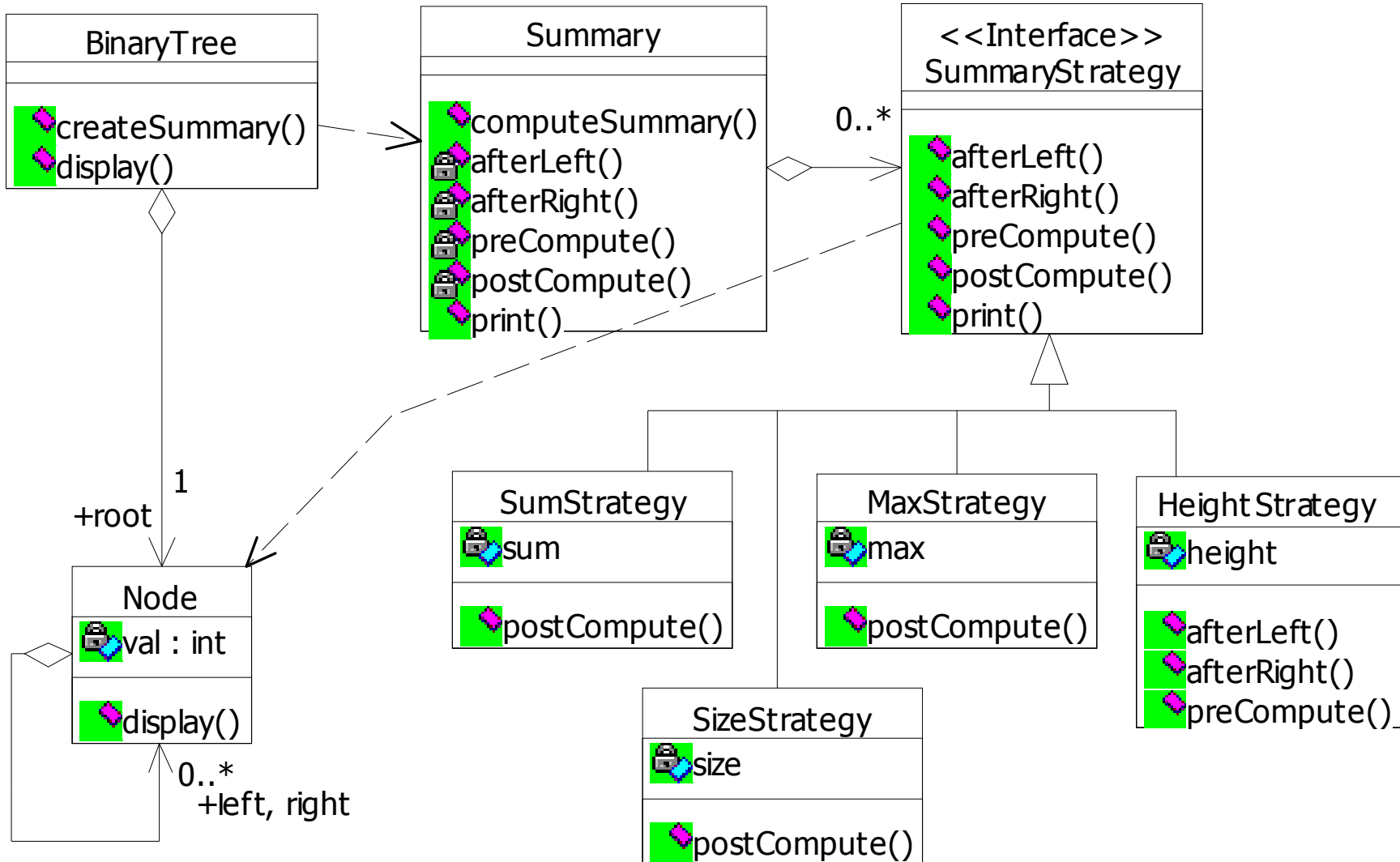
```
Class Summary {
    void computeSummary(Node node) {
        if (node.left != null) {
            computeSummary(node.left);
        }
        if (node.right != null) {
            computeSummary(node.right);
        }
    }
}

class Max extends Summary {
    void computeSummary(Node node) {
        super.computeSummary(summary);
        if (node.val > max)
            max = val;
    }
}
```

# Subclassing (3)



# Strategies



# Strategies (2)

```
public class Summary {
    SummaryStrategy[] strategies;
    public Summary(SummaryStrategy[] strategies) {
        this.strategies = strategies;
    }
    void print() {
        System.out.println("Summary of the tree:");
        for (int i=0; i<strategies.length; i++)
            strategies[i].print();
    }
    void computeSummary(objectModel.Node node) {
        preCompute(node);
        if (node.getLeft() != null) {
            computeSummary(node.getLeft());
            afterLeft(node);
        }
        if (node.getRight() != null) {
            computeSummary(node.getRight());
            afterRight(node);
        }
        postCompute(node);
    }
}
```

# **Aspect-Oriented Programming (AOP)**

A novel programming paradigm that allows clean modularisation of crosscutting concerns, such as debugging and profiling, in software design.

# AO Solution – Feature.Max

```
void computeSummary(Node node) {
    if (node.left != null) {
        computeSummary(node.left);
    }
    if (node.right != null) {
        computeSummary(node.right);
    }
}

package MaxBinaryTree;
class Summary {
    int max;
    void computeSummary(Node node) { // after
        if (node.val > max)
            max = val;
    }
}
```

# AO Solution – Feature.Max (2)

-hyperspace

```
hyperspace BinaryTreeSummaryHyperspace
composable class objectModel.*;
composable class treeSummary.*;
composable class treeSummary.max.*;
```

-concerns

```
package objectModel : Feature.Kernel
package treeSummary : Feature.TreeSummary
package treeSummary.max : Feature.Max
```

-hypermodules

```
hypermodule Max
hyperslices:
```

```
Feature.Kernel,
```

```
Feature.TreeSummary,
```

```
Feature.Max,
```

```
relationships:
```

```
mergeByName;
```

```
end hypermodule;
```

**Hyper/J  
configuration  
file**

# AO Solution – Feature.Height

```
void computeSummary(Node node) {
```

```
    if (node.left != null) {  
        computeSummaryLeft(node);
```

```
    }
```

```
    if (node.right != null) {  
        computeSummaryRight(node);
```

```
    }
```

```
}
```

```
void computeSummaryLeft(Node node) {  
    computeSummary(node.left);
```

```
}
```

```
void computeSummaryRight(Node node) {  
    computeSummary(node.right);
```

```
}
```

# AO Solution – Feature.Height (2)

```
package treeSummary.height;
public class Summary {
    int height;

    void computeSummary(Node node) {
        node.leftHeight = 0;
        height = 0;
    }

    void computeSummaryLeft(Node node) {
        node.leftHeight = ++height;
    }

    void computeSummaryRight(Node node) {
        if (node.leftHeight > ++height)
            height = node.leftHeight;
    }
}
```

# AO Solution – Feature.Height (3)

-concerns

```
package objectModel : Feature.Kernel  
package treeSummary : Feature.TreeSummary  
package treeSummary.height : Feature.Height
```

-hypermodules

hypermodule SumSizeMaxHeight

hyperslices:

Feature.Kernel,

```
Feature.TreeSummary,  
Feature.Height;
```

relationships:

mergeByName;

```
order action Feature.Height.Summary.computeSummary before  
action Feature.TreeSummary.Summary.computeSummary;
```

end hypermodule;

## Hyper/J configuration file

# AO Solution Vs. OO

	Scattered	Tangled	Run-time overhead	Code size overhead
#ifdef	++	++		
Subclassing			+	++
Strategy			+	+
AO			+	+