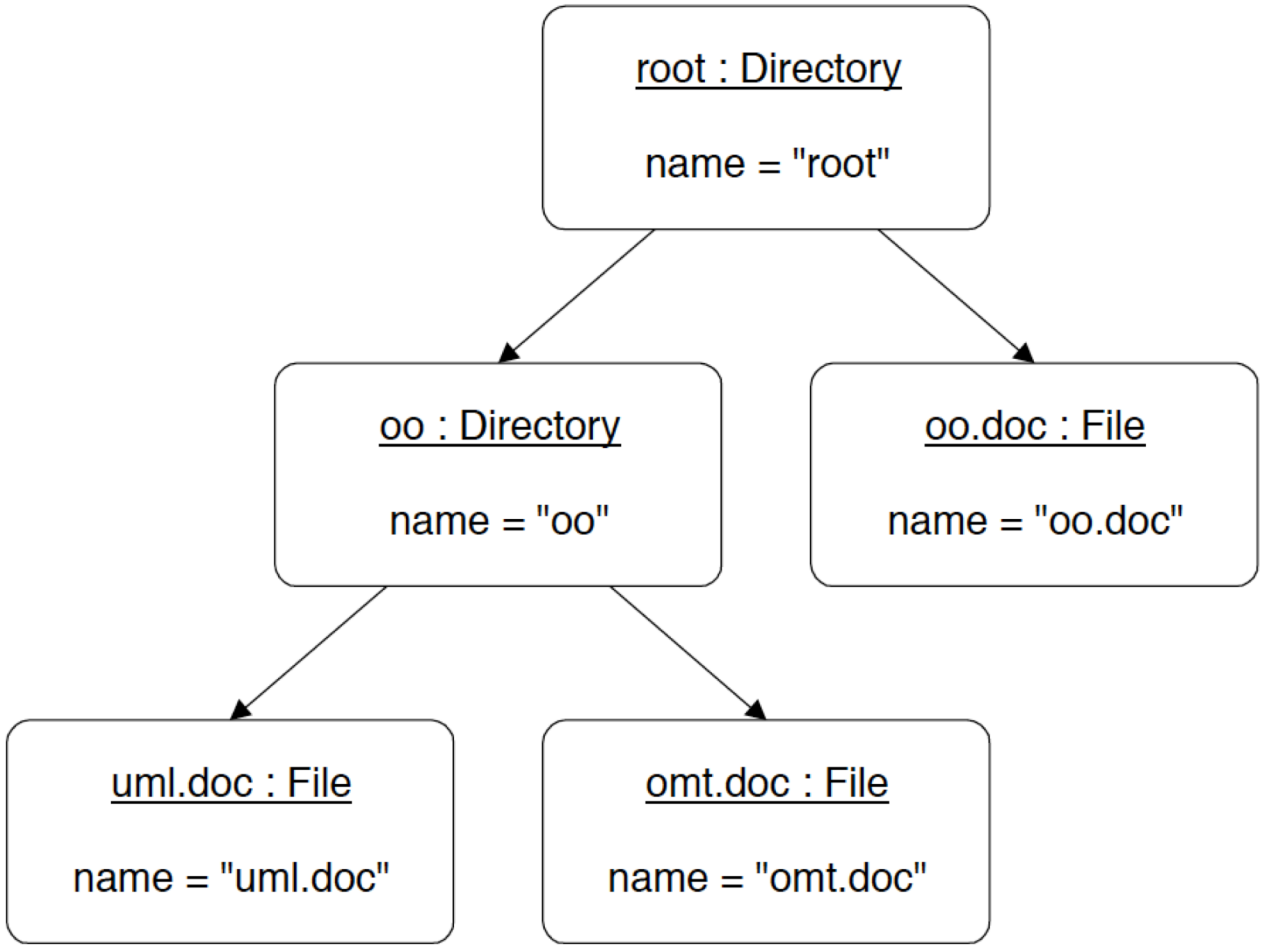
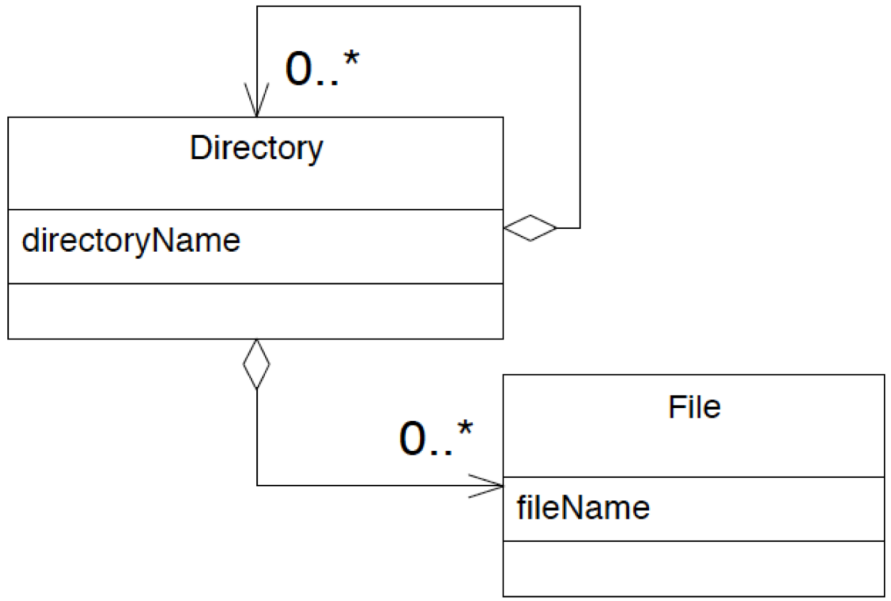
Composite Design Pattern

Recursive Aggregation



Recursive Aggregation (cont.)

- This is how we often model *object hierarchies*, such as a the directory hierarchy for a file system.
- Sometimes we use the ***Composite*** Design Pattern instead...



Design Pattern: *Composite*

Intent: Model a tree-like hierarchy, such that branches and leaves can be manipulated uniformly.

Implementation: Usually *recursive*.

Applicability:

- Any hierarchical organization of objects & compositions of objects.

Pros:

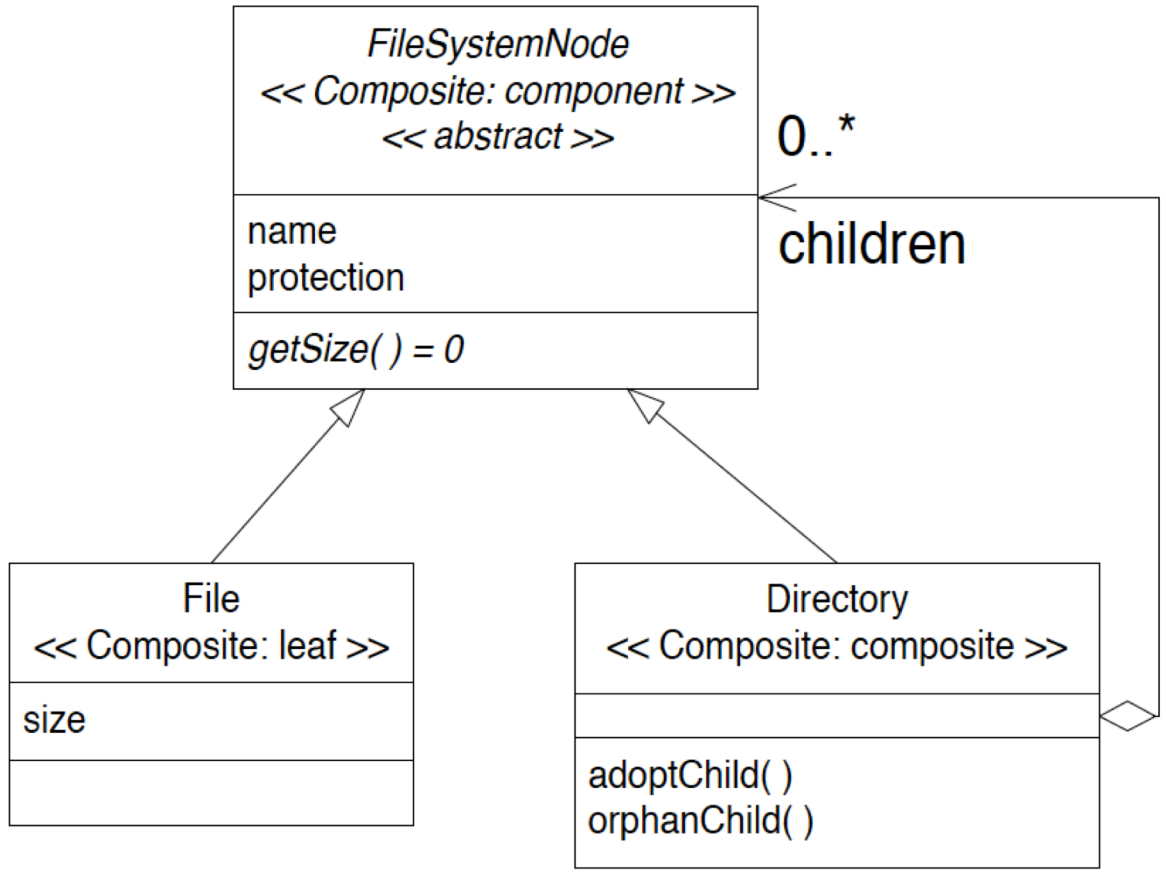
- Simplifies client code whenever clients don't know or care whether a given node (component) is a leaf or a branch.
- It is easy to add new component types.

Cons:

- Whenever the client *does* care about the type of a given node, it must implement run-time type-checks and down-casts; otherwise, compile-time checks would have been sufficient.

Composite Example

- Write pseudo-code for the getSize() method(s).

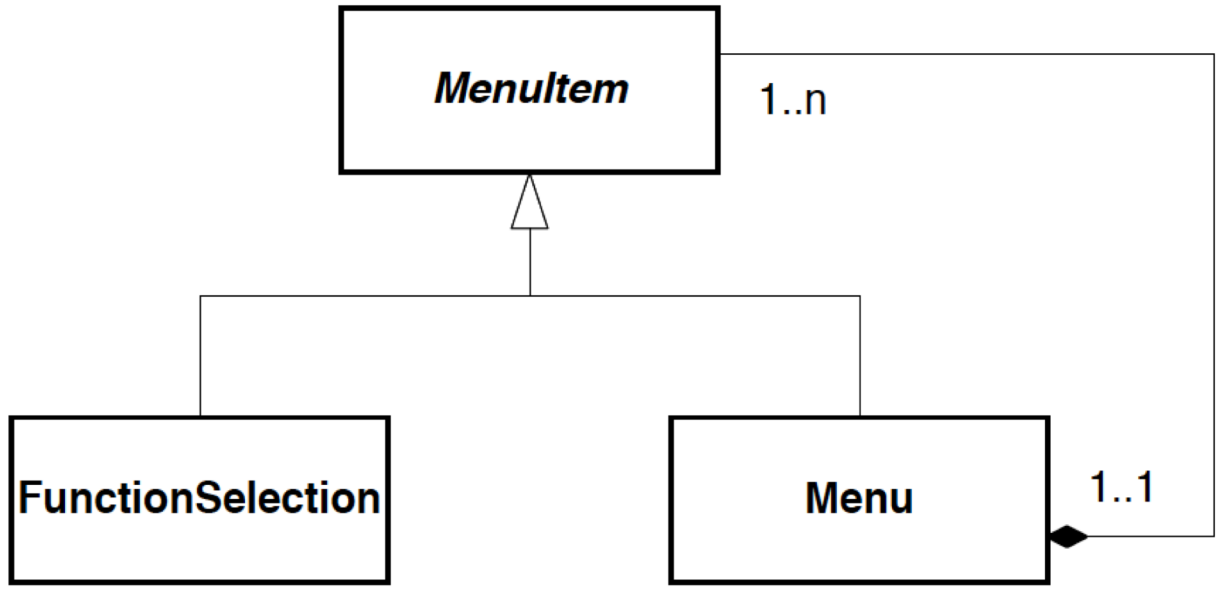


Composite Example Code

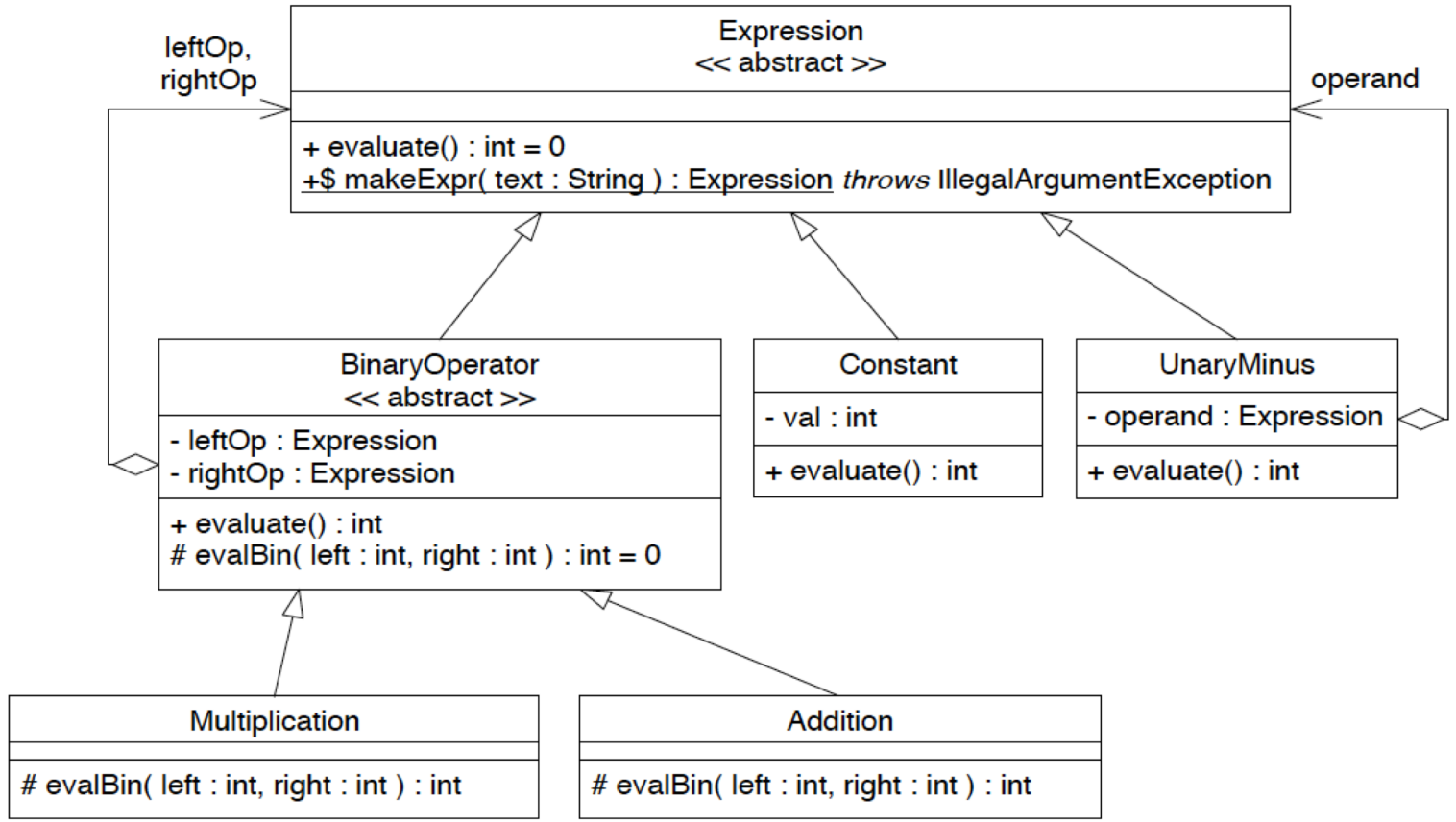
```
class Directory extends FileSystemNode {
 public int getSize() {
    int size = 0;
    FileSystemNode child = getFirstChild();
    while( child ) {
       size += child.getSize(); // Recursive & Polymorphic
       child = getNextChild();
    }
    return size;
}...}
class File extends FileSystemNode {
public int getSize() {
    return size;
}...}
```

Another Composite Example

- UI Menus are composed of menu selections.
- A selection can:
  - Invoke any program function.
  - Bring up another menu.

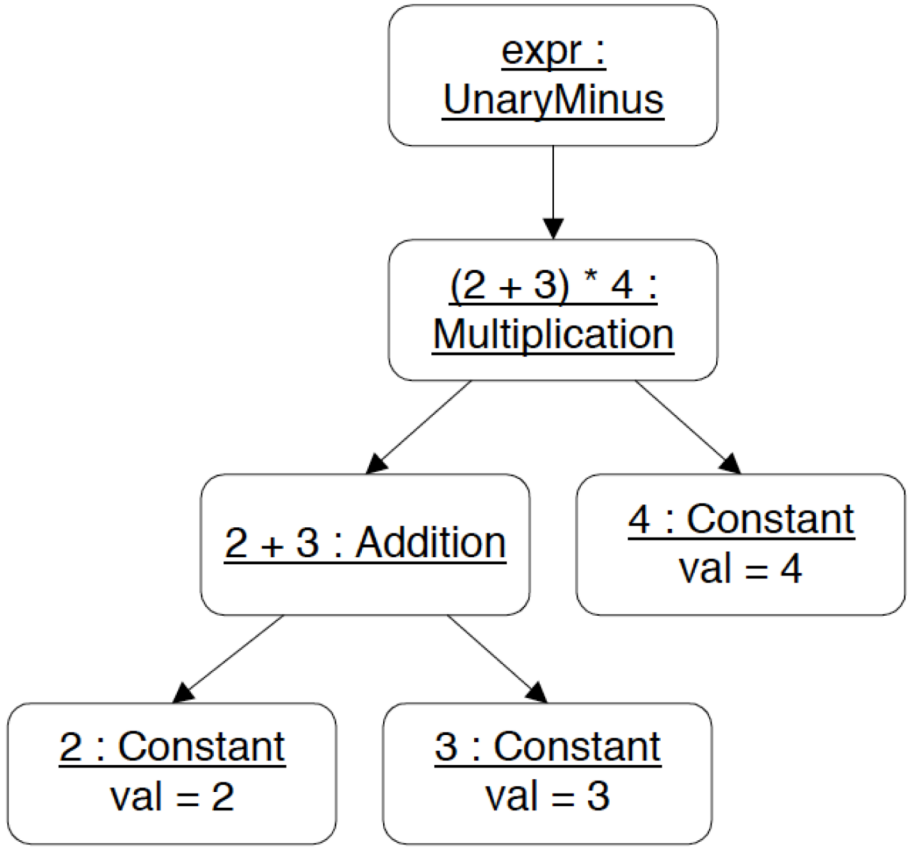


Composite Example: Expressions

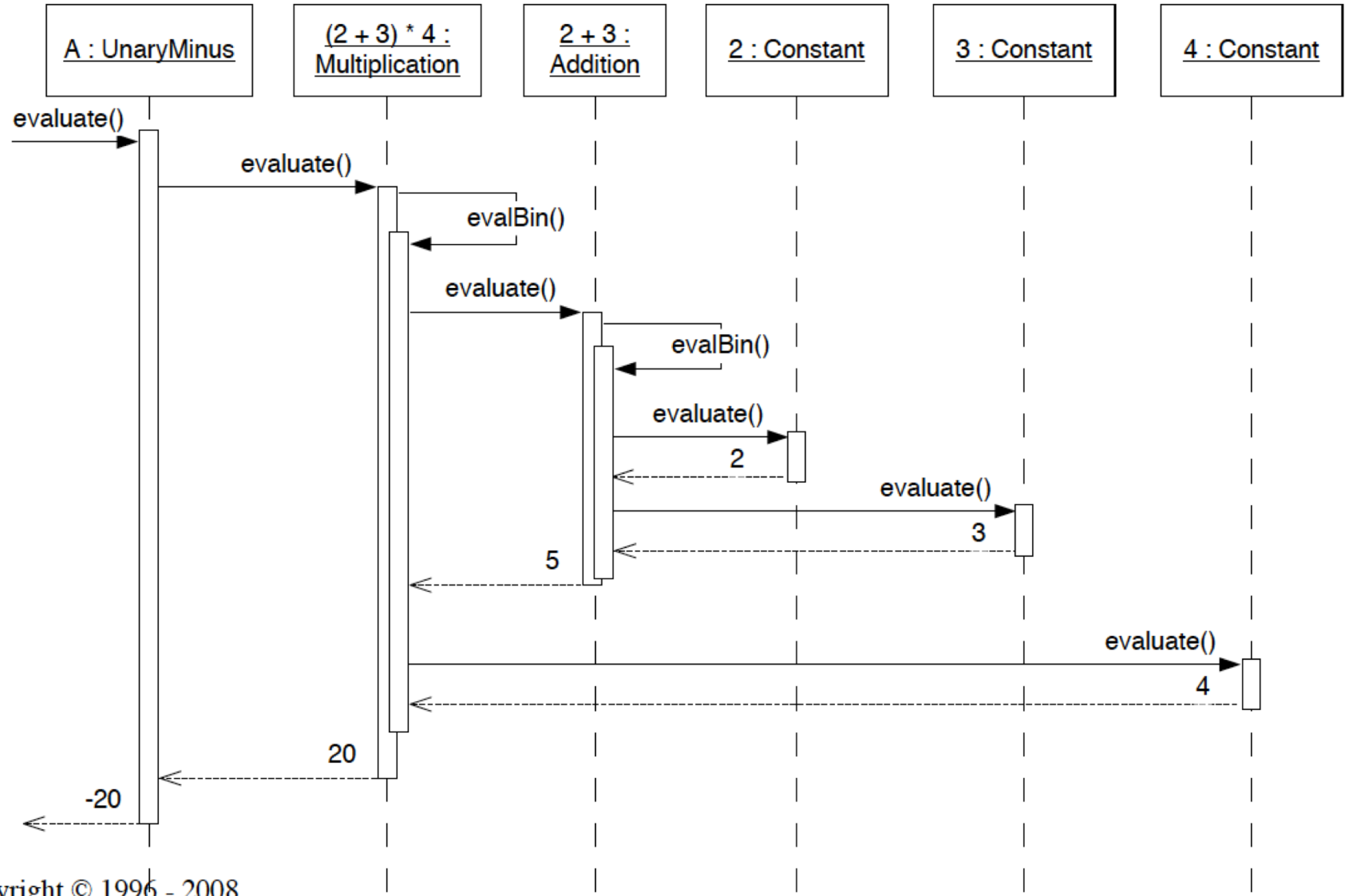


Composite Example

`expr = -((2+3)*4)` *evaluates to* **`-20`**



Composite Expressions



Composite Expressions (cont.)

```
// Ignore Constructors. . .
abstract class Expression {
  public abstract int evaluate();
  public Expression makeExpr( String text )
    throws IllegalArgumentException { // Virtual Constructor
} }
class Constant extends Expression {
  private int val;
  public int evaluate() {
    return val;
} }
class UnaryMinus extends Expression {
  private Expression operand;
  public int evaluate() {
    return - operand.evaluate();
} }
```

Composite Expressions (cont.)

```
abstract class BinaryOperator extends Expression {
  private Expression leftOp;
  private Expression rightOp;
  protected abstract int evalBin( int left, int right );
  public int evaluate() {
    // Template Method
    return evalBin( leftOp.evaluate(), rightOp.evaluate() );
} }
class Addition extends BinaryOperator {
  protected int evalBin( int left, int right ) {
    return left + right;
} }
class Multiplication extends BinaryOperator {
  protected int evalBin( int left, int right ) {
    return left * right;
} }
```

A simple application

Keeps track of parts in a manufacturing environment and how they are use in constructing assemblies. We'll use the following terminology:

Parts Individual, physical objects.

Assemblies Complex structures made up out of parts, and possibly structured into *subassemblies*.

The program could:

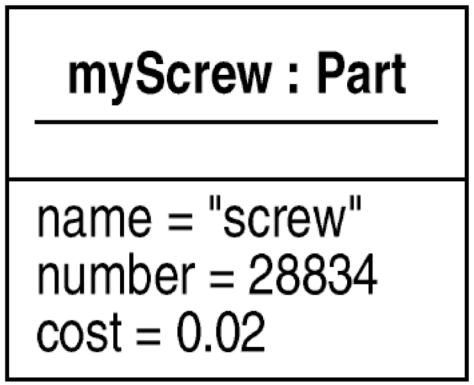
- maintain catalogue information: what kinds of parts are there?
- maintain inventory information: how many are in stock?
- record structure of manufactured assemblies.
- provide operations on assemblies:
  - cost, based on cost of component parts;
  - print “parts explosion”.

A major task of object-oriented design is: *to decide how a system's data should be split up between different objects*.

Rule of thumb

Consider application domain concepts, such as *part*, as candidate objects. What data do we need to store about parts?

- An identification number
- A description
- Their price
- …

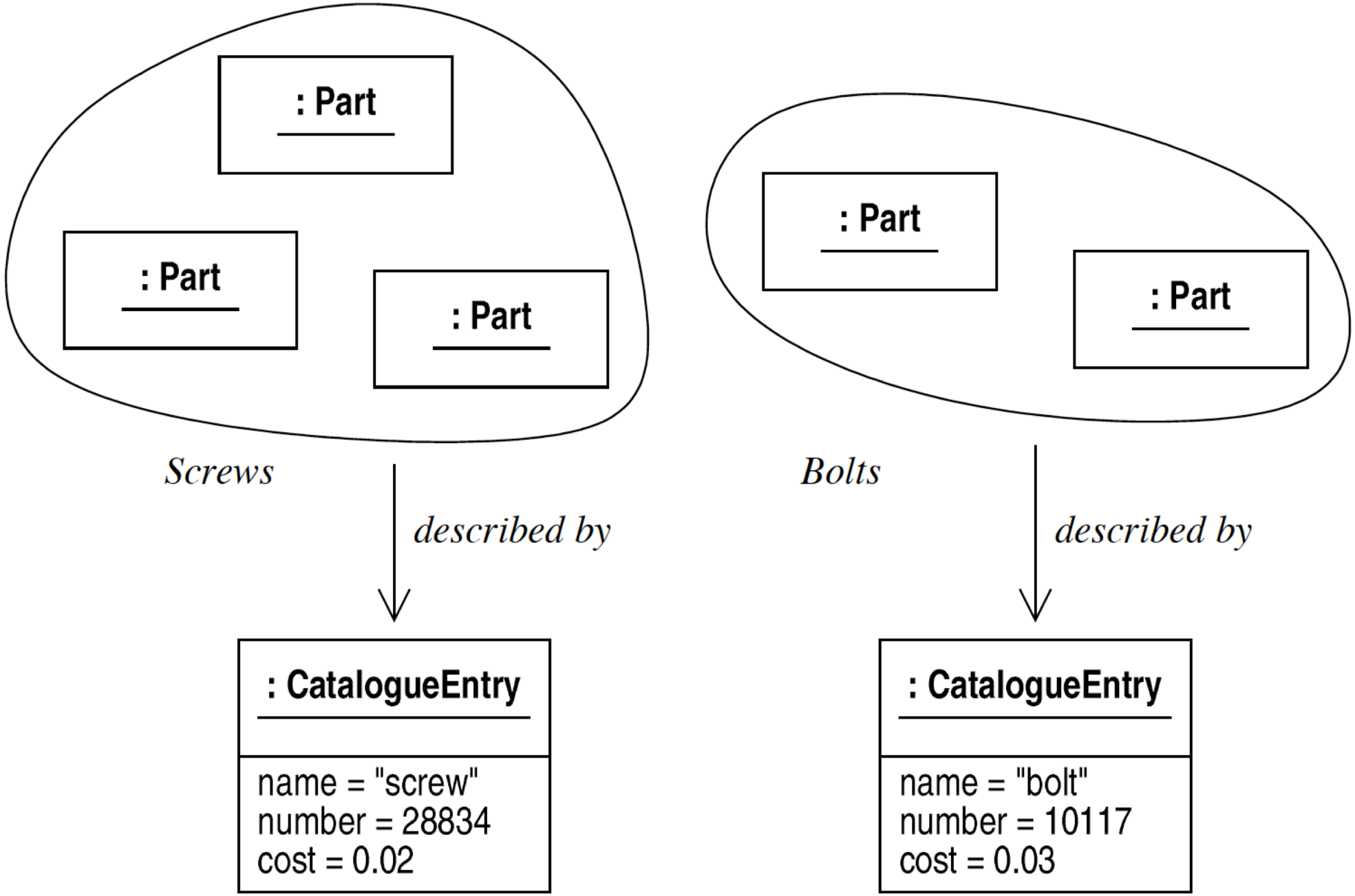


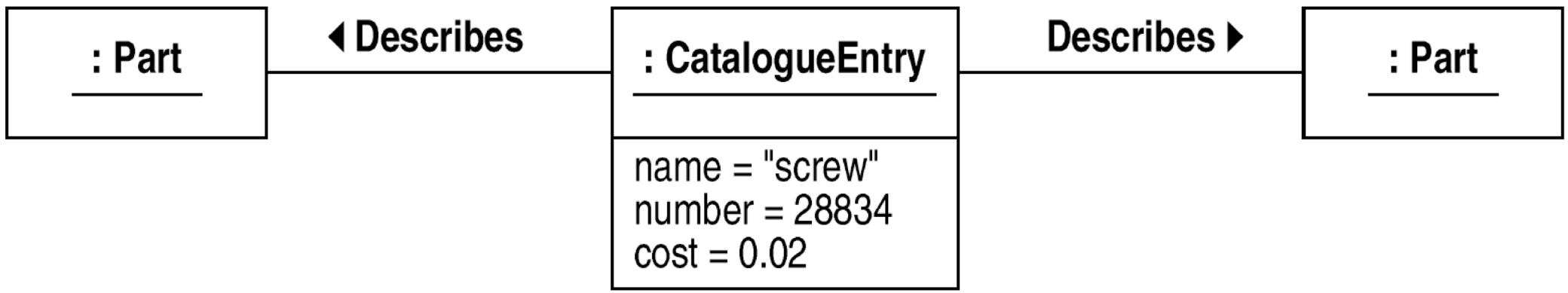
If we have 100,000 screws, this design would represent them by 100,000 screw objects, each storing the ID number of the part, its description and its price.

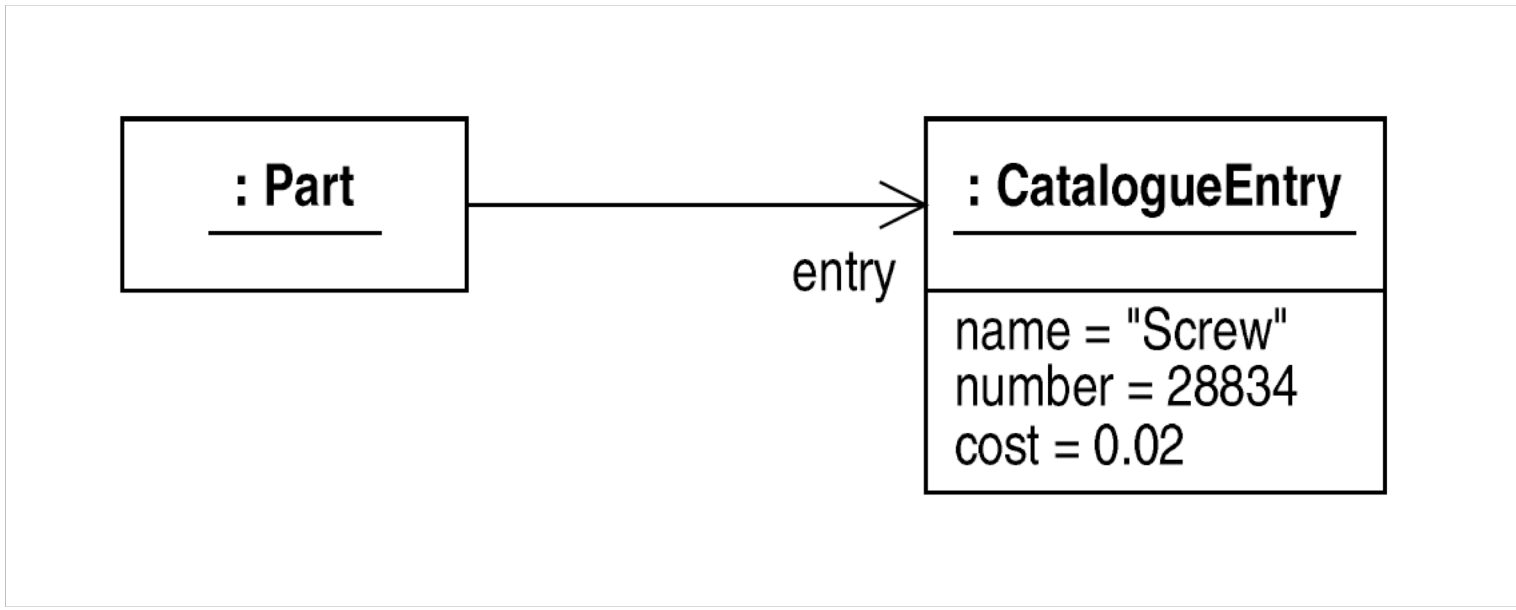
There are at least two major problems with this approach:

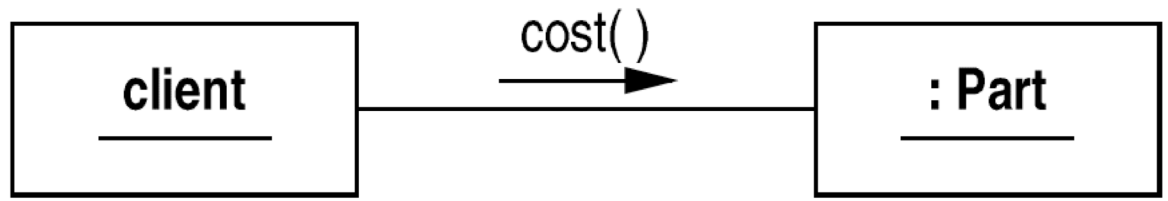
Redundancy Storing the same data 100,000 times will waste significant amounts of storage.

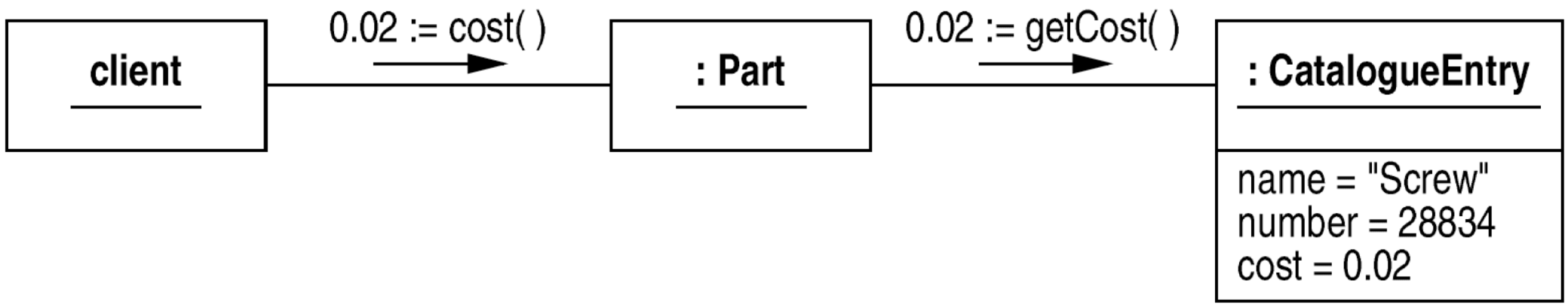
Maintainability If for example the price of a screw changed, there would be 100,000 updates to perform in order to update the system. As well as being time-consuming, this will make it very hard to ensure that the system's data is accurate and consistent.


: Part
: Part
: Part
Screws
described by
: CatalogueEntry
name = "screw"
number = 28834
cost = 0.02
: Part
: Part
Bolts
described by
: CatalogueEntry
name = "bolt"
number = 10117
cost = 0.03

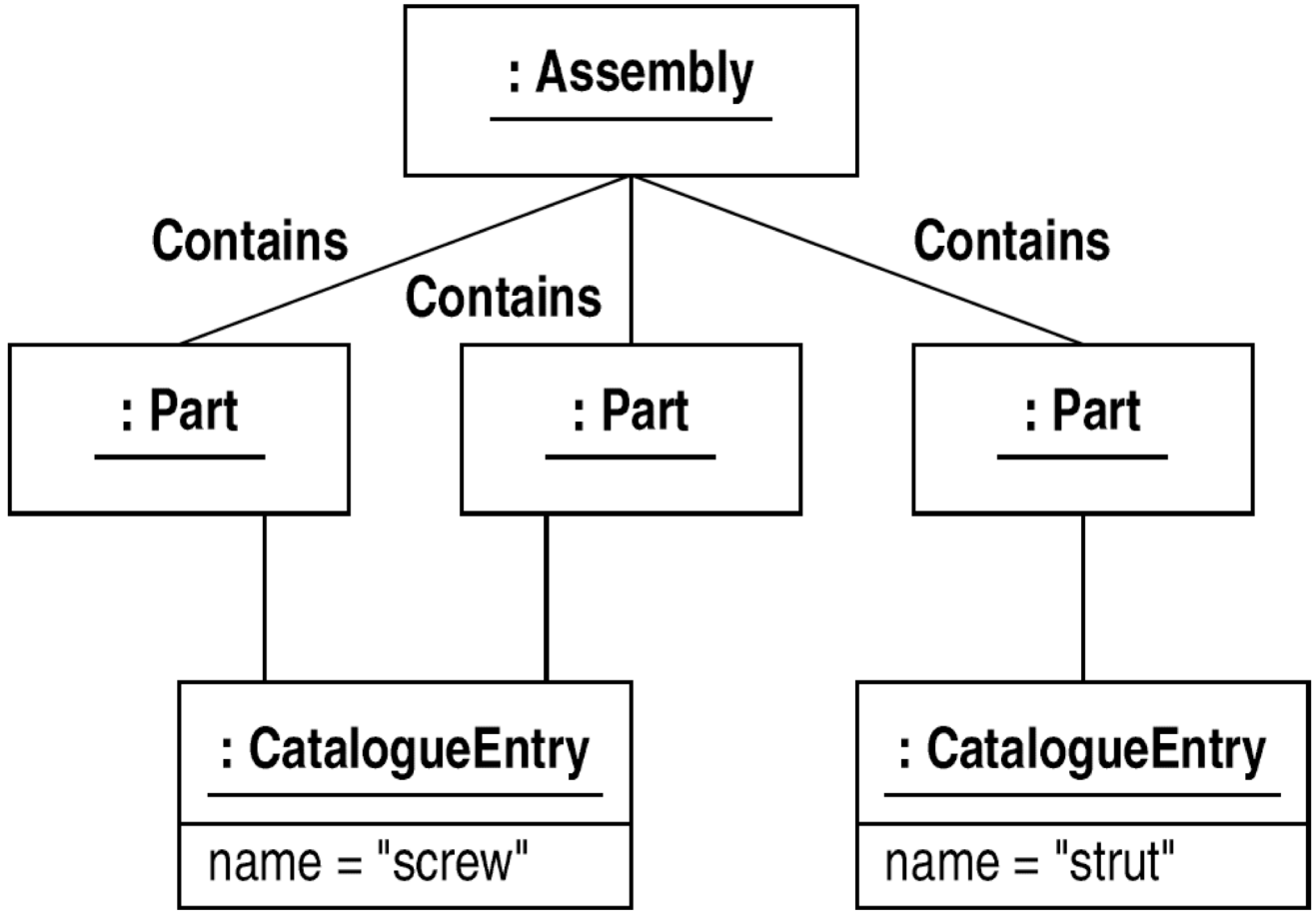

: Part
Describes
: CatalogueEntry
name = "screw"
number = 28834
cost = 0.02
Describes
: Part


: Part
: CatalogueEntry
entry
name = "Screw"
number = 28834
cost = 0.02


client
cost( )
: Part


client
0.02 := cost( )
: Part
0.02 := getCost( )
: CatalogueEntry
name = "Screw"
number = 28834
cost = 0.02

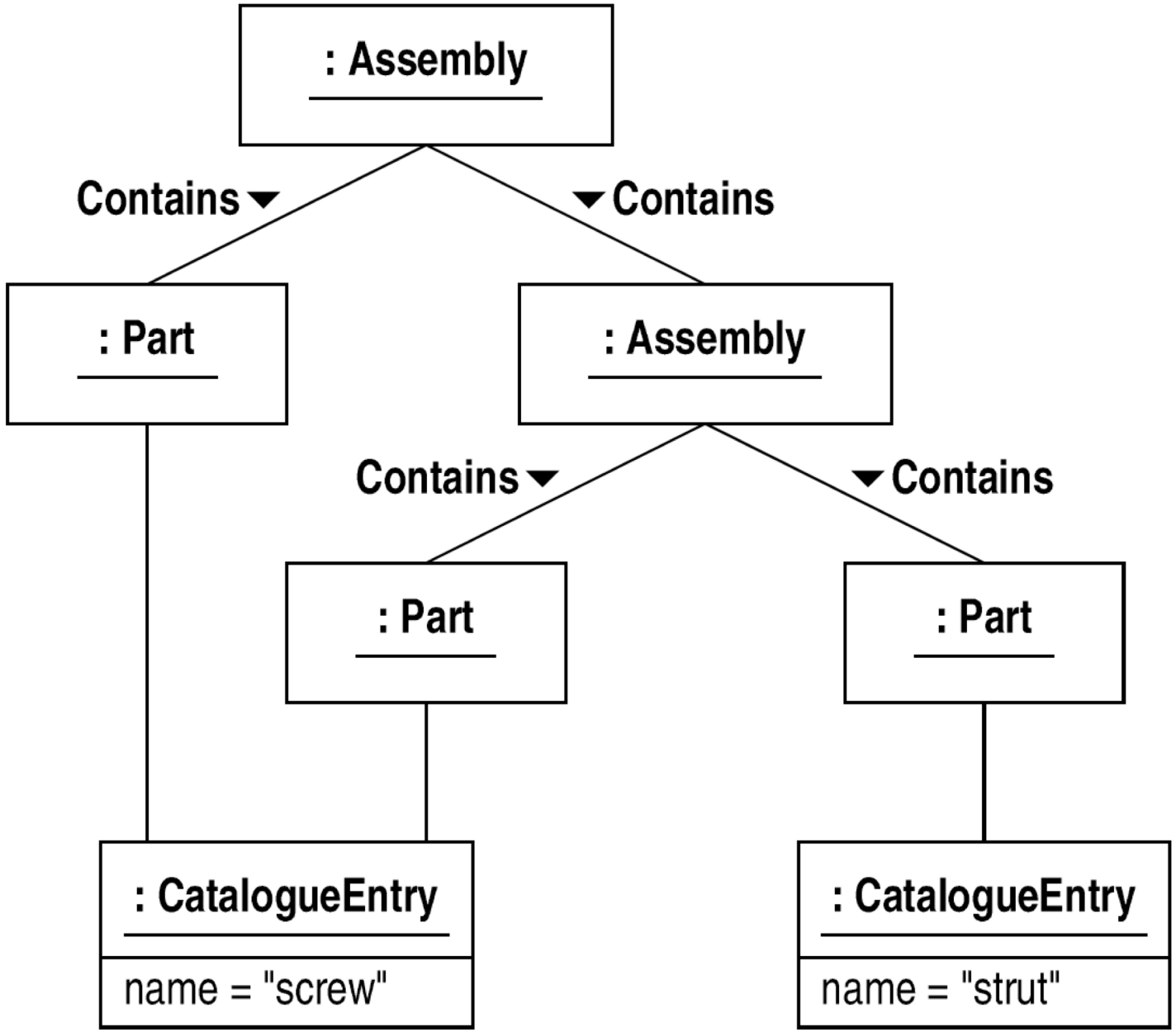
An assembly is a grouping of individual parts that together make up a structure.



All the information about the parts in the assembly is held implicity, represented by the links connecting part objects to the assembly.

An implementation of the assembly class must contain:

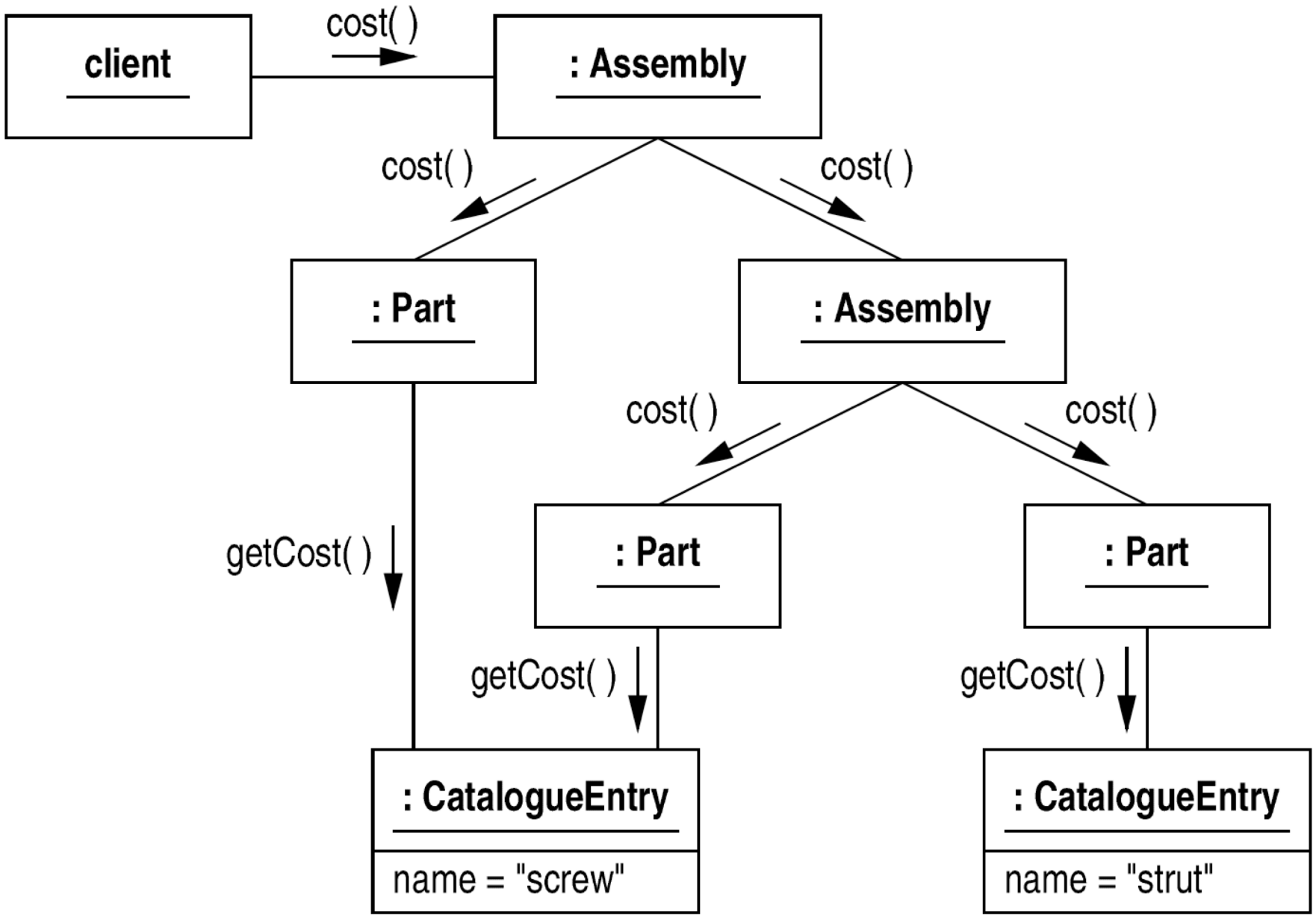
1. a data structure to store references to linked objects
2. “housekeeping” operations to add, test and remove links from this collection

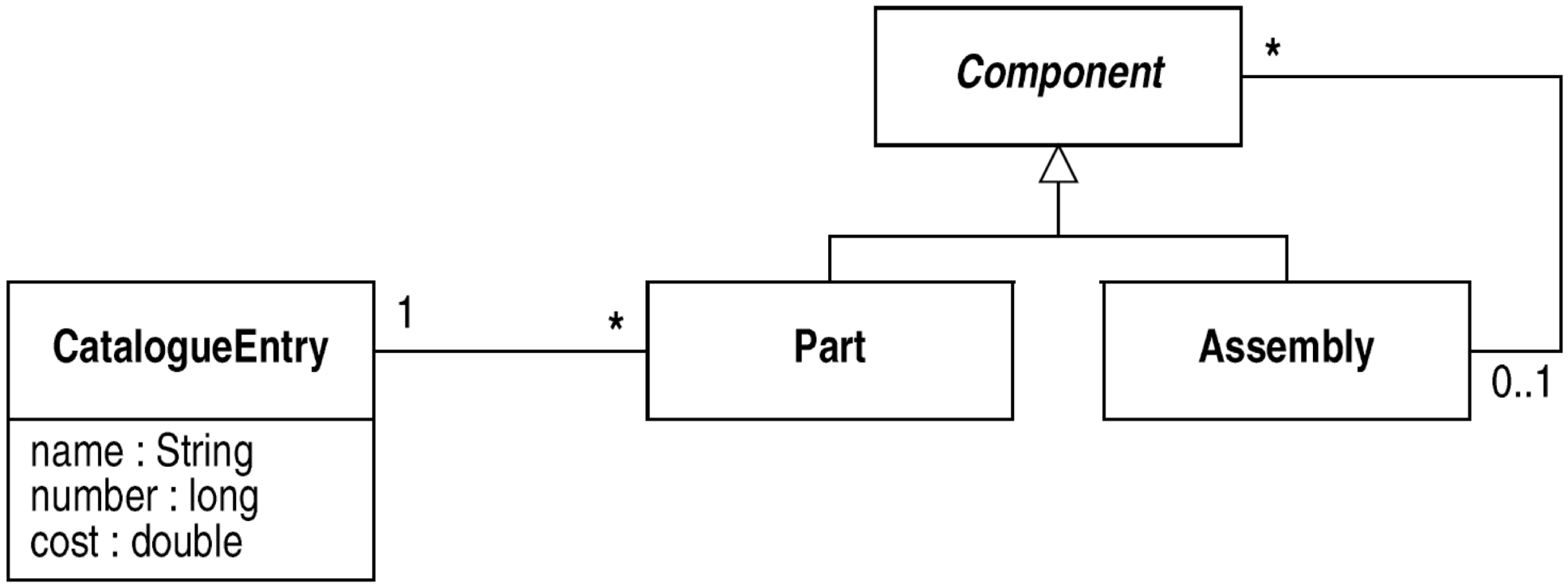

: Assembly
Contains
Contains
: Part
: Assembly
Contains
Contains
: Part
: Part
: CatalogueEntry
name = "screw"
: CatalogueEntry
name = "strut"

No objects of the component class are ever created. It exists only to describe the common features of parts and assemblies, namely

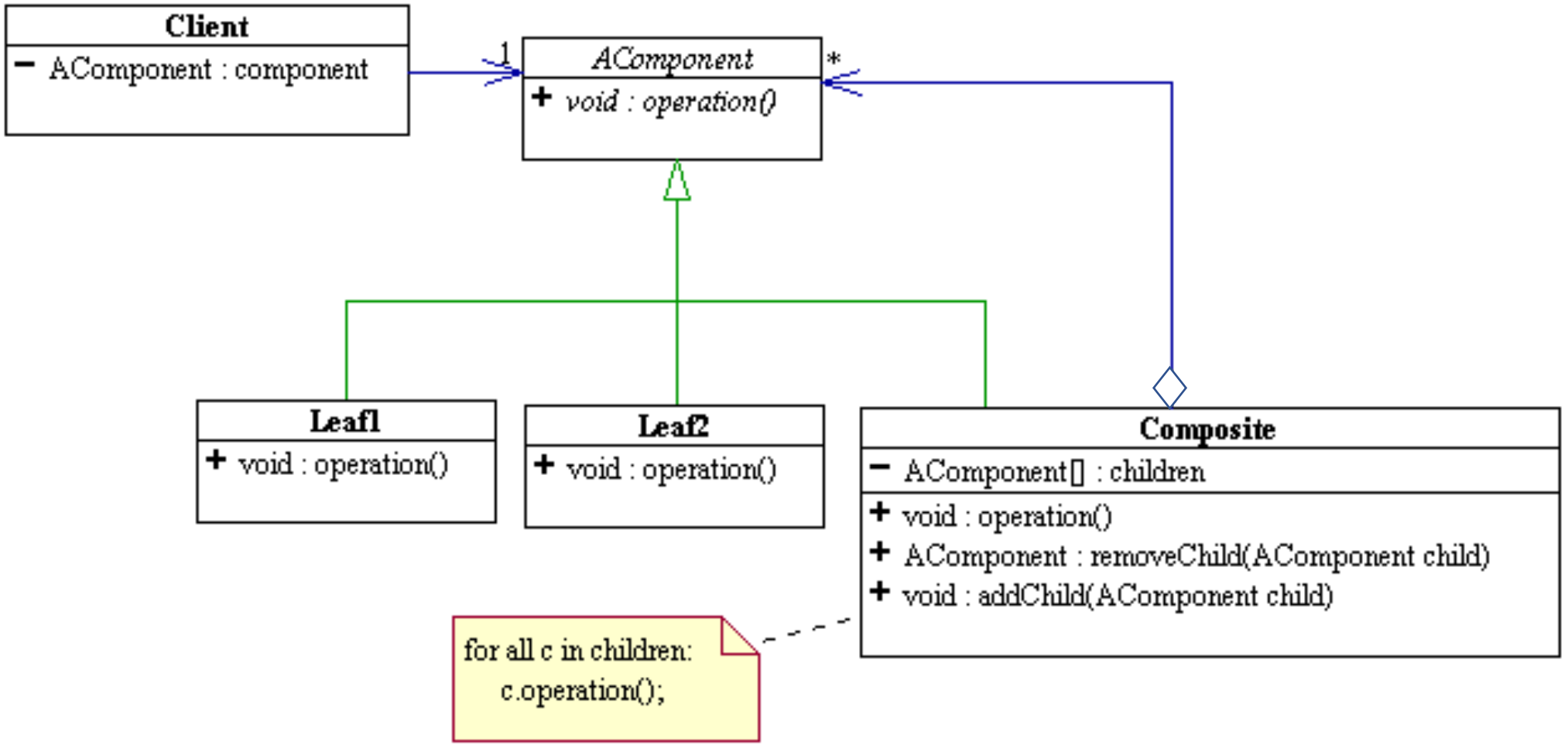
- that they can both be stored in assemblies, and
- that we can find out the cost of both.

Such a class is called an *abstract* class.

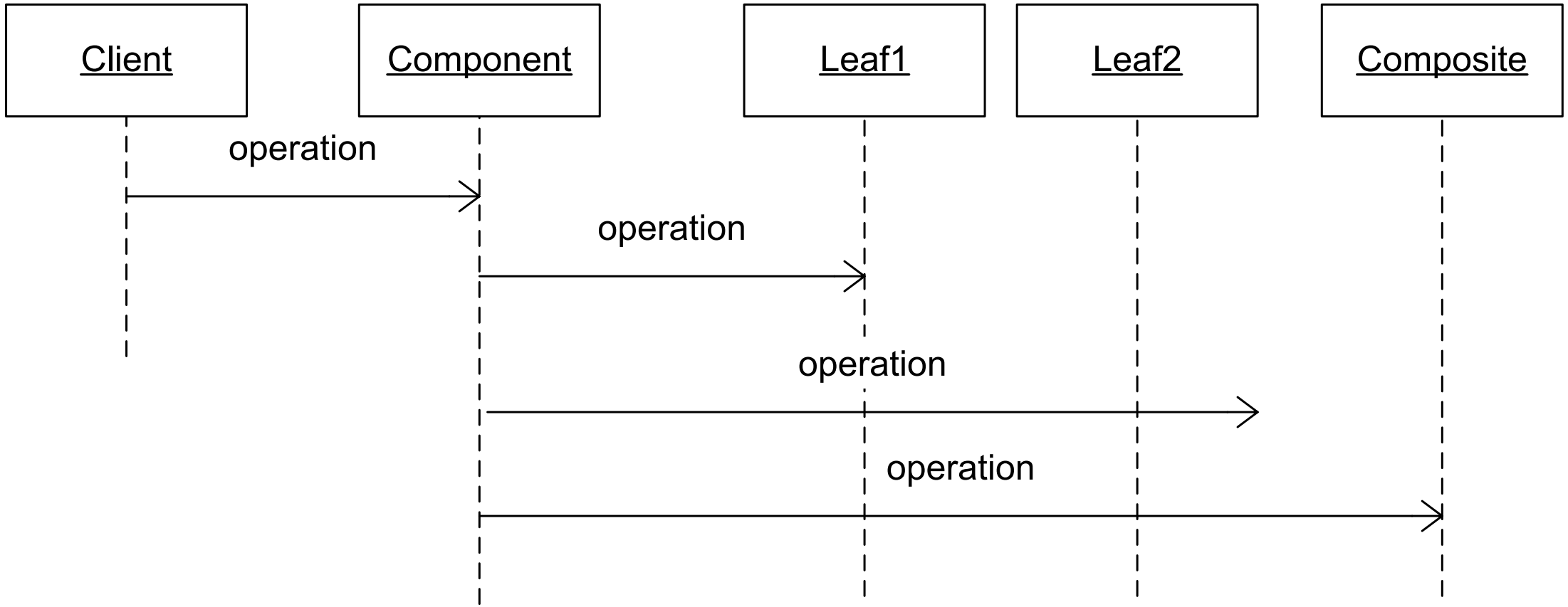

client
cost( )
: Assembly
cost( )
cost( )
: Part
: Assembly
cost( )
cost( )
getCost( )
: Part
: Part
getCost( )
getCost( )
: CatalogueEntry
name = "screw"
: CatalogueEntry
name = "strut"


Component
*
CatalogueEntry
name : String
number : long
cost : double
1
*
Part
Assembly
0..1

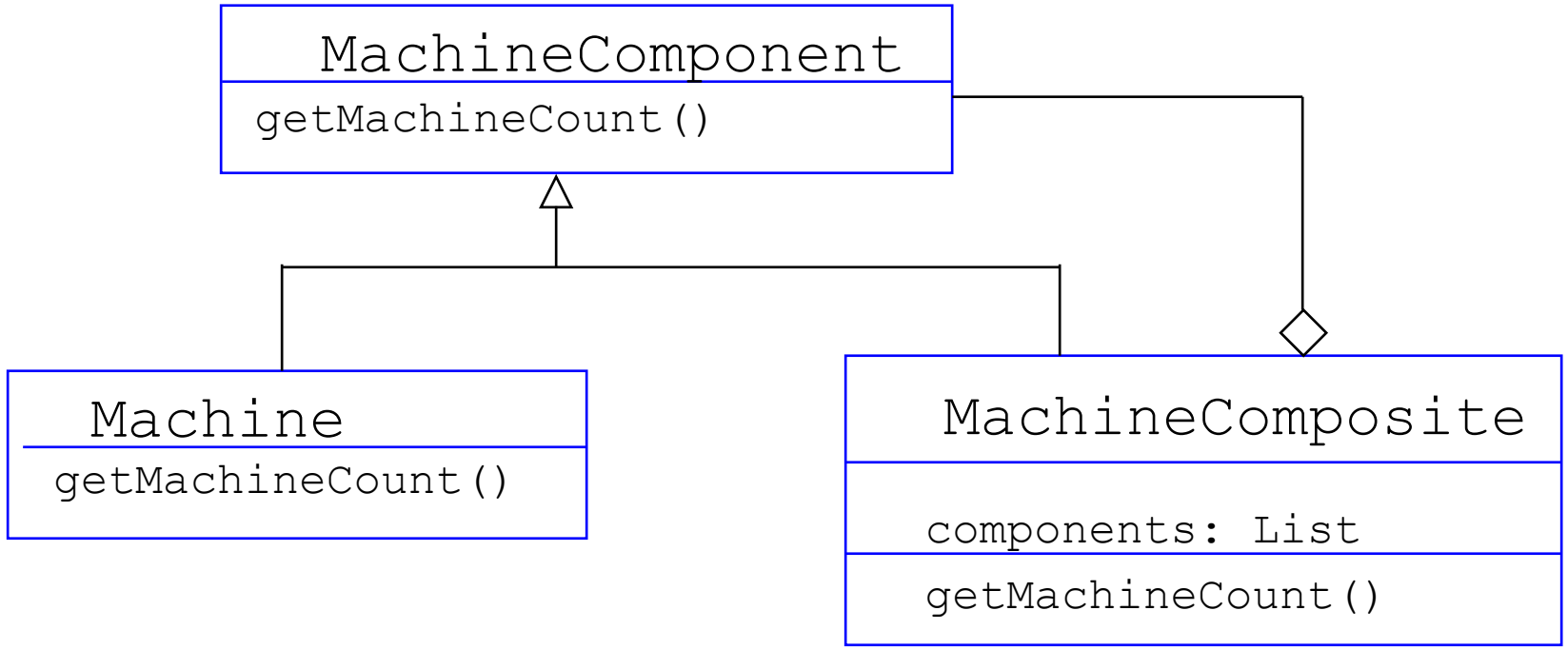
Composite Design Pattern: Abstract Description



Composite Design Pattern: Abstract Description



Composite Design Pattern: Concrete Example



getMachineCount() returns the number of machines in any given component (Machine returns 1; MachineComposite returns sum of the counts for each component)

Composite Design Pattern

- Consequences
  - Recursive data structures lends itself to recursive processing
  - Simplifies the client
    - No concern for looping on composites
  - Explicit hierarchy makes it easier to add new leaf of composite types

Composite Design Pattern

- **Implementation Issues**
  - Typically necessary to have multiple types of composites
    - Numbers and roles of composite parts
  - Composite should protect its internal data structure
    - If necessary, give client iterator over component parts
  - Maximize the abstract interface for client isolation
    - Provide all methods of all derived classes
    - Isolates client from needing to type cast to get what they need

Composite Design Pattern

- **Implementation Issues**
  - Where should we declare the "child management" methods?
    - For example – addChild, removeChild, …
    - Option I : in the Component class
      - » again, isolates client from distinction
      - » costs safety, meaningless to add to leaf
      - » can add default behavior for all composites
    - Option II: in the Composite class
      - » Safe – only defined on composites
      - » Creates different interfaces for leaf and composites
      - » Can provide "isComposite" to help with safe type casting

Composite Design Pattern

- **Implementation Issues**
  - Don't declare "child list" in Component
    - tempting if child management introduced at this level
    - costs space in all leaf nodes
  - Pointers to parents
    - Makes certain kinds of traversal easier

Composite Design Pattern

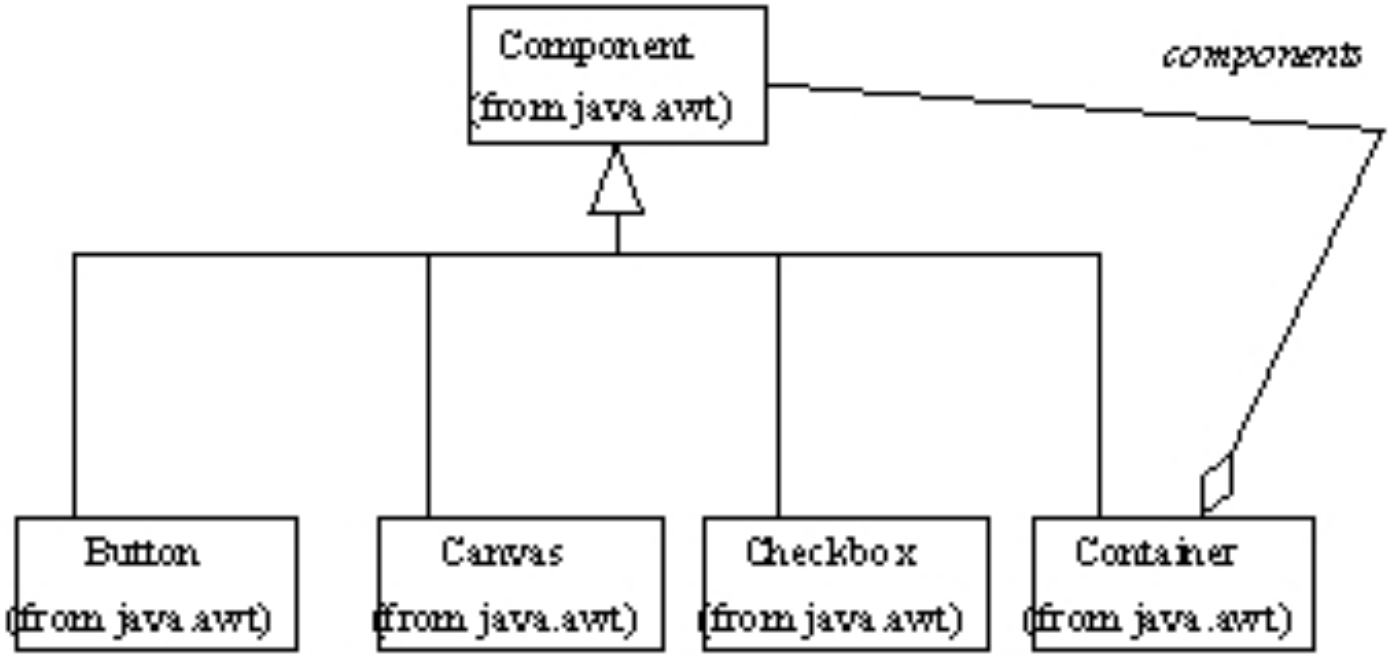
- Example: Shapes
  - Could support Shapes built from other shapes
  - Sample Code

```
public abstract class Shape {
  . . .
}
public class CompositeShape {
  protected Vector parts;
  public void addPart(Shape s){…}
  public void draw(Graphics g){
    for(int i=0 ; i<parts.size(); i++){
      ((Shape)(parts.get(i))).draw(g);
    }
. . .
```

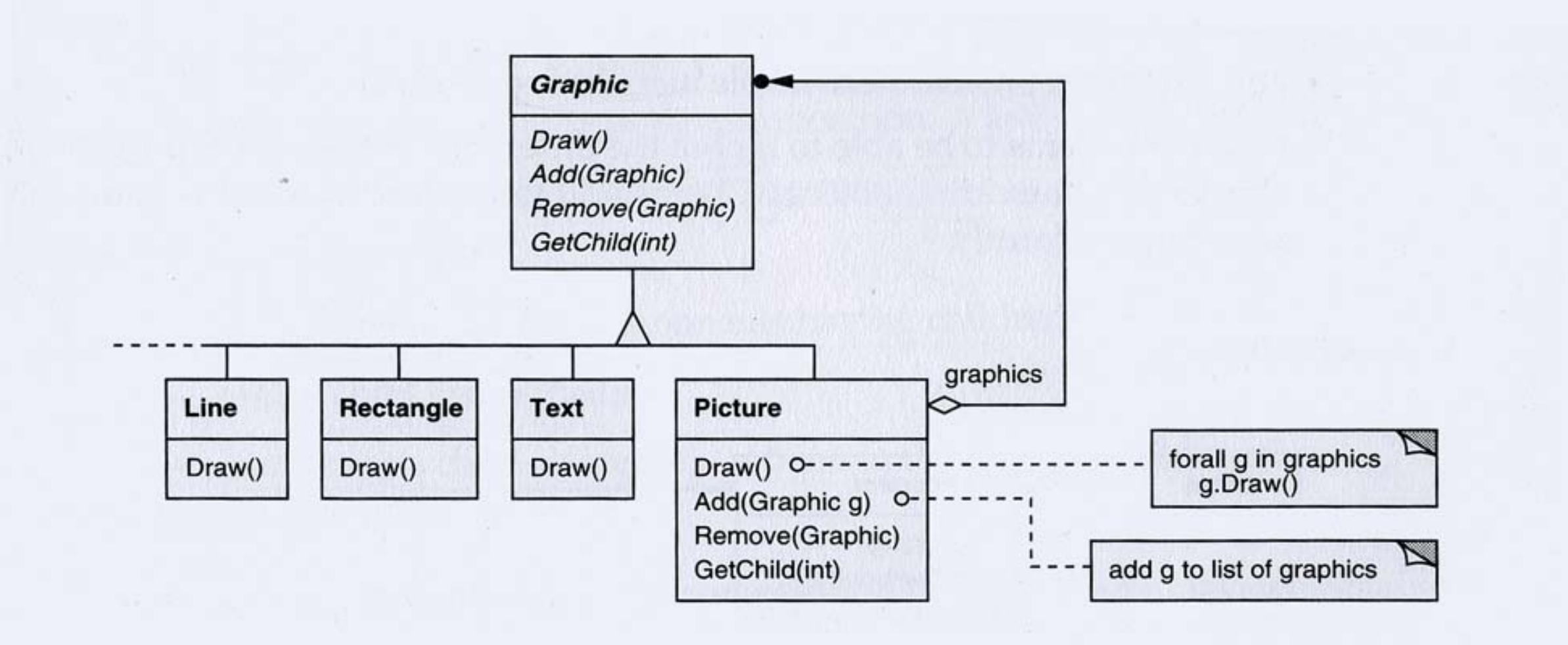
Composite Design Pattern

- Java API Usage

  java.awt.Component



Composite Design Pattern for Pictures



Observations

- The Component (Graphic) is an abstract class that declares the interface for the objects in the pattern. As the interface, it declares methods (such as Draw) that are specific to the graphical objects.
- Line, Rectangle, and Text are so-called Leafs, which are subclasses that implement Draw to draw lines, rectangles, and text, respectively.
- The Picture class represents a number of graphics objects. It can call Draw on its children and also uses children to compose pictures using primitive objects.

Considerations

- The Composite Pattern is used to represent part-whole object hierarchies.
- Clients interact with objects through the component class.
- It enables clients to to ignore the specifics of which leaf or composite class they use.
- Can be used recursively, so that Display can show both flares and stars.
- New components can easily be added to a design.

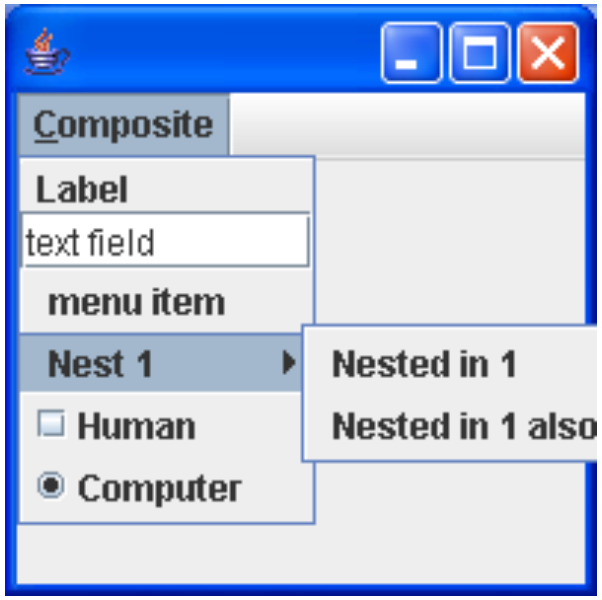
Use Example: Java Swing

- Java Swing has four major pieces:
  - Events and EventListeners
  - Layouts
  - Drawing
  - Graphical Components
    - The root of all of these is named: Component
- Component utilizes the Composite pattern in several ways, with menus for example
  - One you may find useful or need for your projects

JMenus in Java Swing

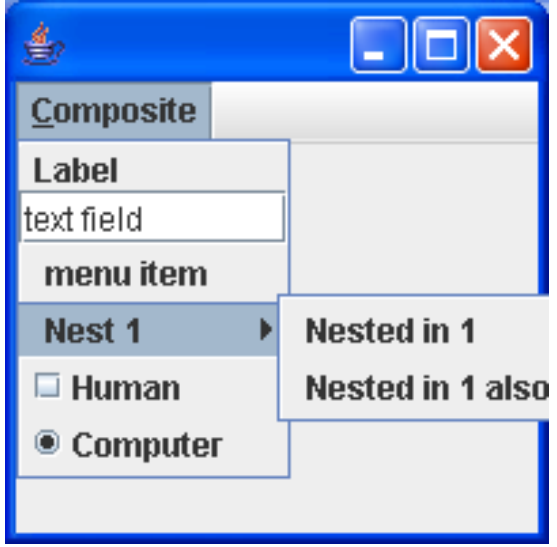
- Java menus use the Composite Design Pattern
- **`JMenuBar`** is a Composite extending **`JComponent`**
- **`JMenuBar`** is a Composite extending **`JComponent`**
  - Can add others like **`JLabel`**, **`JTextField`**
  - Can also add **`JMenuItem`** to **`JMenuItem`**
- **`JMenuItem`** has three subclasses
  - **`JMenu`**
  - **`JRadioButtonMenuItem`**
  - **`JCheckboxMenuItem`**

```
JMenuItem menu = new JMenu("Composite");
menu.setMnemonic('C');//Open with alt-C
// Create two leafs
JLabel label = new JLabel("Label");
JTextField textF = new JTextField("text field");
menu.add(label);
menu.add(textF);
// Add a Composite
JMenuItem menuItem = new JMenuItem("menu item");
menu.add(menuItem);
// Add two Composites to a Composite
JMenuItem jmi1Nest = new JMenu("Nest 1");
menu.add(jmi1Nest);
JMenuItem jmiNested1 = new JMenuItem("Nested in 1");
jmi1Nest.add(jmiNested1);
JMenuItem jmiNested2 = new JMenuItem("Nested in 1 also");
jmi1Nest.add(jmiNested2);
```



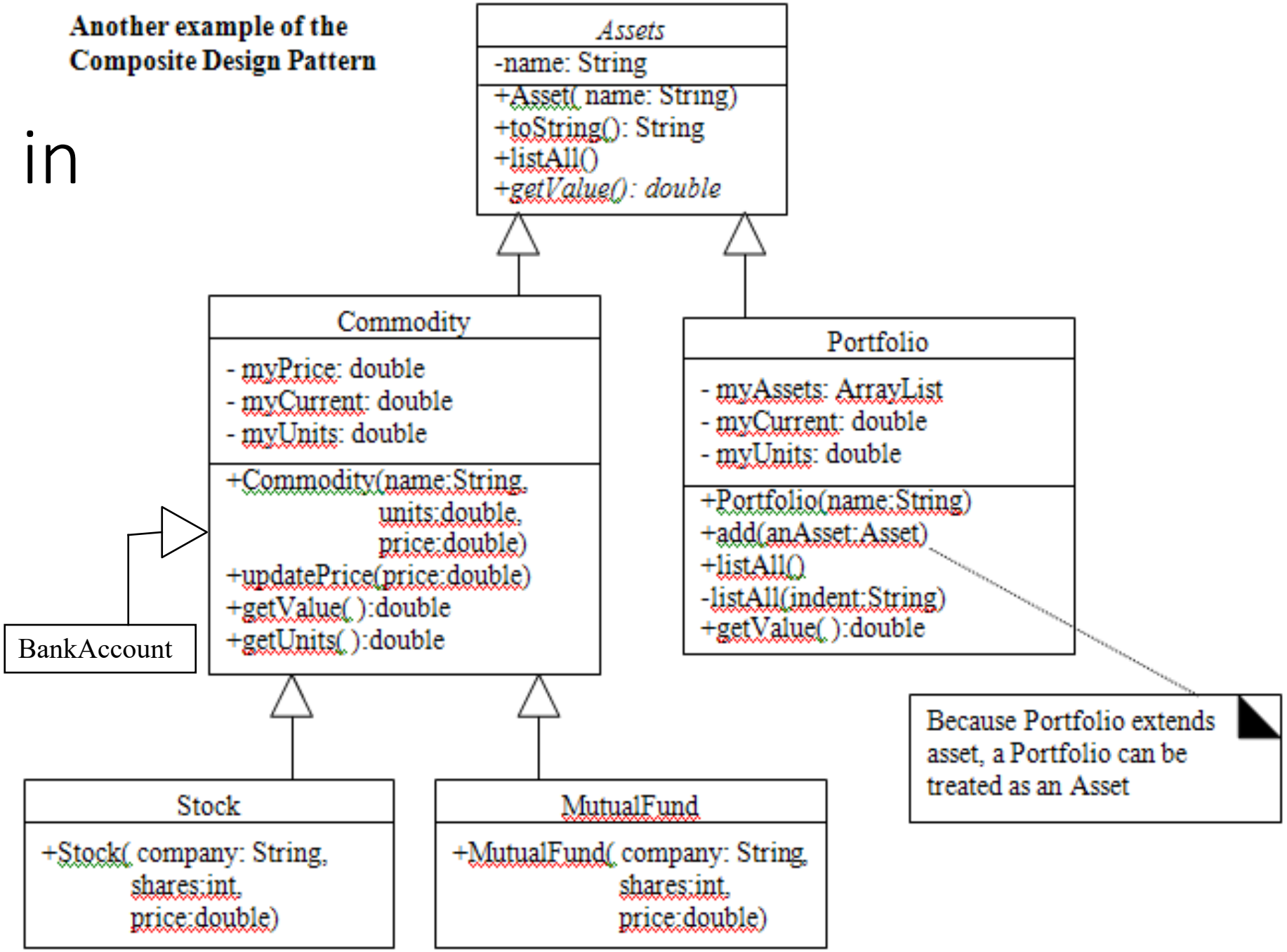
JMenuItemDemoComposite

```
// Add two more Composites
JMenuItem checkBox
     = new JCheckBoxMenuItem("Human", false);
JMenuItem radioButton
     = new JRadioButtonMenuItem("Computer", true);
menu.add(checkBox);
menu.add(radioButton);
// Add two more Composites
JMenuBar menuBar = new JMenuBar();
setJMenuBar(menuBar);
menuBar.add(menu);
```



Run JMenuItemDemoComposite.java

Portfolios in Portfolios



```
Portfolio overall = new Portfolio("My IRA");

    // Add two leafs to "my IRA"
    Stock aStock = new Stock("Seven Eleven", 500, 9.15);
    overall.add(aStock);
    BankAccount account = new BankAccount("Swiss Account", 300000);
    overall.add(account);

    // Create a tech portfolio
    Portfolio tech = new Portfolio("Tech Stocks");
      Stock oracle = new Stock(”Oracle", 20, 30.50);
      MutualFund highRisk = new MutualFund("Nasdaq", 13, 45.20);

      tech.add(oracle); // add leaf
      tech.add(highRisk); // add leaf
      // Add this 2nd portfolio to the overall portfolio
      overall.add(tech); // add Composite
      // overall.add(overall); There is an if to avoid adding this to this

      // Create an overseas portfolio of tech stocks
      Portfolio global = new Portfolio("Global Equities");
        Stock pacificRim = new Stock("Pacific Rim Tech", 10, 12.34);
        MutualFund lrgGrow = new MutualFund("Large Growth", 100, 95.21);
        global.add(pacificRim);
        global.add(lrgGrow);
        tech.add(global);
```

Recursive toString result

My Entire Retirement Portfolio with value $315417.0
  500 shares of Seven Eleven with value $4575.0
  BankAccount: 'Swiss Account' with value $300000.0
  Tech Stocks with value $10842.0
    20 shares of Sun with value $610.0
    13.0 shares of Nasdaq with value $587.6
    Global Equities with value $9644.4
      10 shares of Pacific Rim Tech with value $123.4
      100.0 shares of Large Growth with value $9521.0