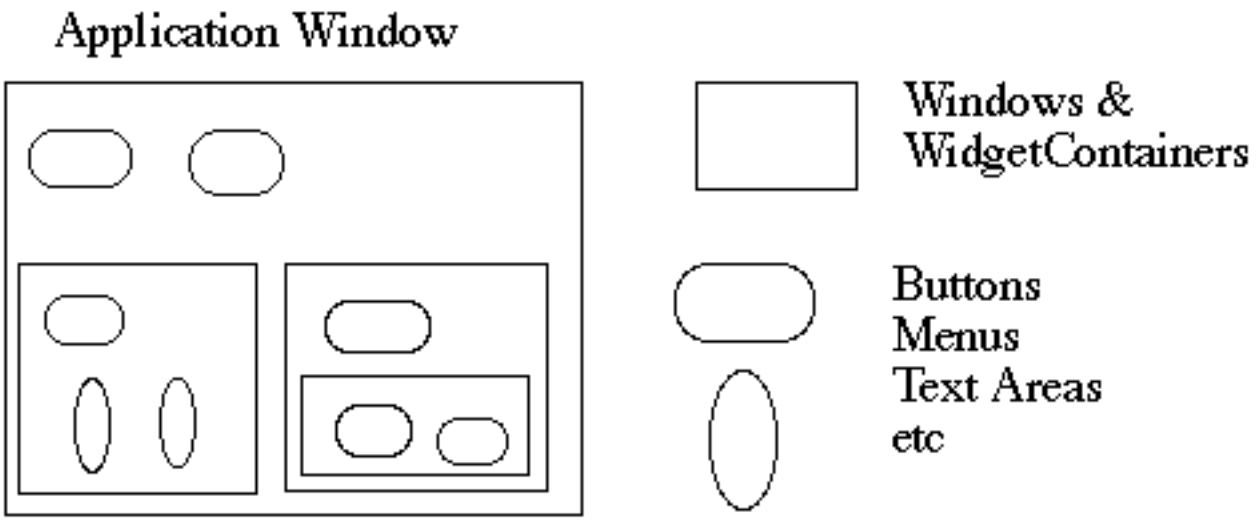
CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2001
Doc 17 Composite & Interpreter
Contents

Reference

Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides, 1995, pp. 163-174, 243-256

Composite
Example - Motivation
GUI Windows and GUI elements



How does the window hold and deal with the different items it has to manage?

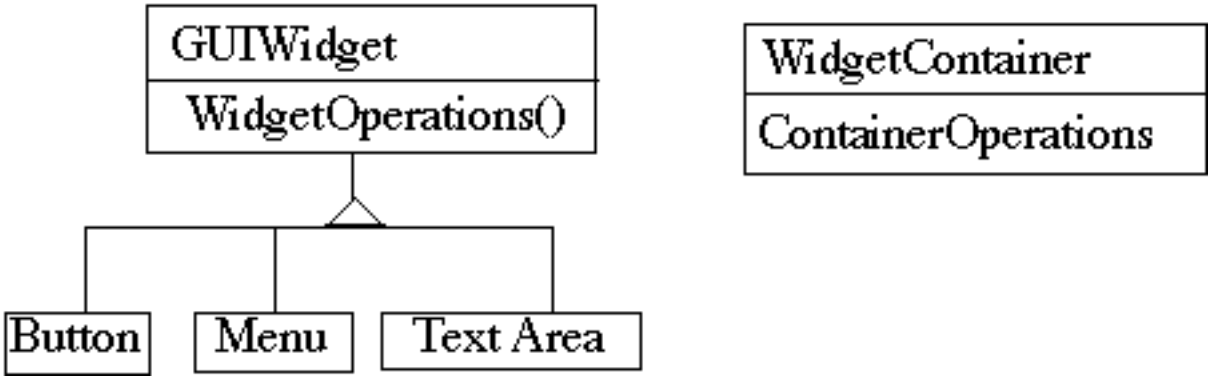
Widgets are different that WidgetContainers

Bad News Implementation

```
class Window
   {
   Buttons[] myButtons;
   Menus[] myMenus;
   TextAreas[] myTextAreas;
   WidgetContainer[] myContainers;

   public void update()
      {
      if ( myButtons != null )
         for ( int k = 0; k < myButtons.length(); k++ )
            myButtons[k].refresh();
      if ( myMenus != null )
         for ( int k = 0; k < myMenus.length(); k++ )
            myMenus[k].display();
      if ( myTextAreas != null )
         for ( int k = 0; k < myButtons.length(); k++ )
            myTextAreas[k].refresh();
      if ( myContainers != null )
         for ( int k = 0; k < myContainers.length(); k++ )
            myContainers[k].updateElements();
      etc.
      }
   public void fooOperation()
      {
      if ( blah ) etc.
      }
   }
```

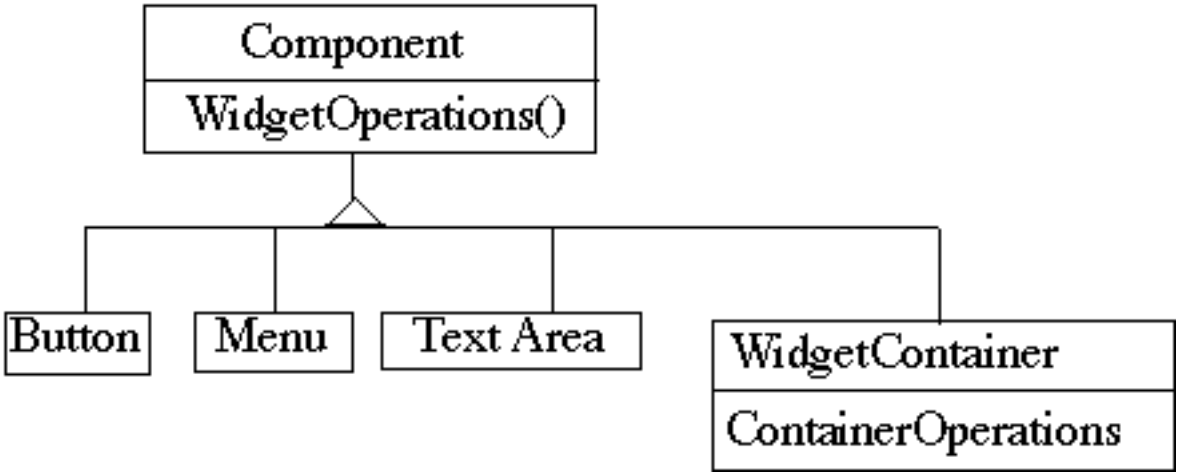
A Better Idea - Program to an interface




```
class Window
   {
   GUIWidgets[] myWidgets;
   WidgetContainer[] myContainers;

   public void update()
      {
      if ( myWidgets != null )
         for ( int k = 0; k < myWidgets.length(); k++ )
            myWidgets[k].update();
      if ( myContainers != null )
         for ( int k = 0; k < myContainers.length(); k++ )
            myContainers[k].updateElements();
      etc.
      }
   }
```

The Composite Pattern



Component implements default behavior for widgets when possible

Button, Menu, etc overrides Component methods when needed

WidgetContainer will have to overrides all widgetOperations

```
class WidgetContainer
  {
  Component[] myComponents;

  public void update()
    {
    if ( myComponents != null )
      for ( int k = 0; k < myComponents.length(); k++ )
        myComponents[k].update();
    }
  }
```

Issue: WidgetContainer Operations

WidgetContainer operations tend to relate to adding, deleting and managing widgets

Should the WidgetContainer operations be declared in Component?

Pro - Transparency

Declaring them in the Component gives all subclasses the same interface

All subclasses can be treated alike. (?)

Con - Safety

Declaring them in WidgetContainer is safer

Adding or removing widgets to non-WidgetContainers is an error

What should be the proper response to adding a TextArea to a button? Throw an exception?

One out is to check the type of the object before using a WidgetContainer operation

Explicit Parent References

Aid in traversing the structure

```
class WidgetContainer
   {
   Component[] myComponents;

   public void update()
      {
      if ( myComponents != null )
         for ( int k = 0; k < myComponents.length(); k++ )
            myComponents[k].update();
      }
   public add( Component aComponent )
      {
      myComponents.append( aComponent );
      aComponent.setParent( this );
      }
   }

class Button extends Component
   {
   private Component parent;
   public void setParent( Component myParent)
      {
      parent = myParent;
      }

   etc.
   }
```

More Issues

Should Component implement a list of Components?

- The button etc. will have a useless data member

Child ordering is important in some cases

Who should delete components?

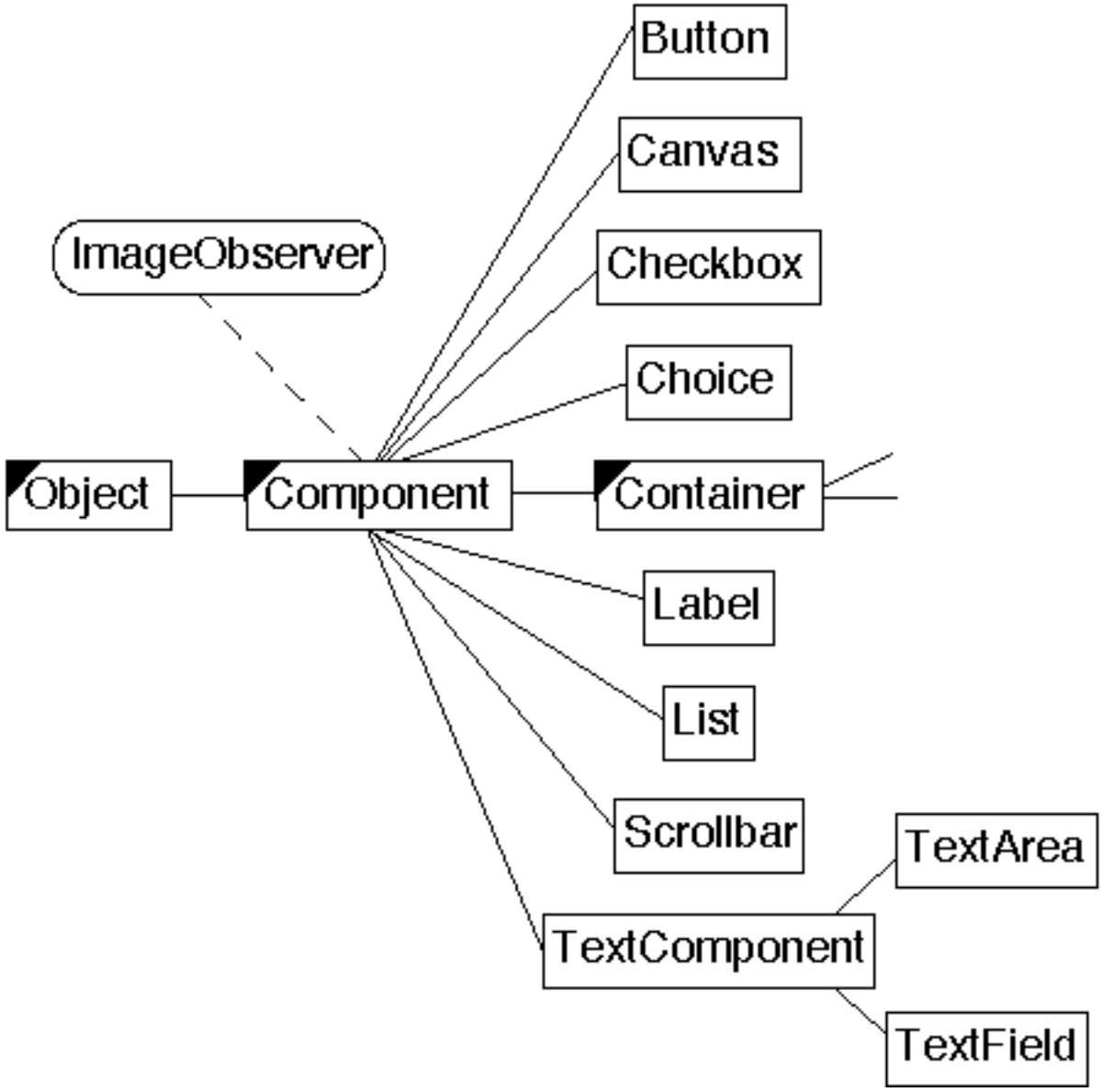
- If there is no garbage collection Container is best bet

Applicability

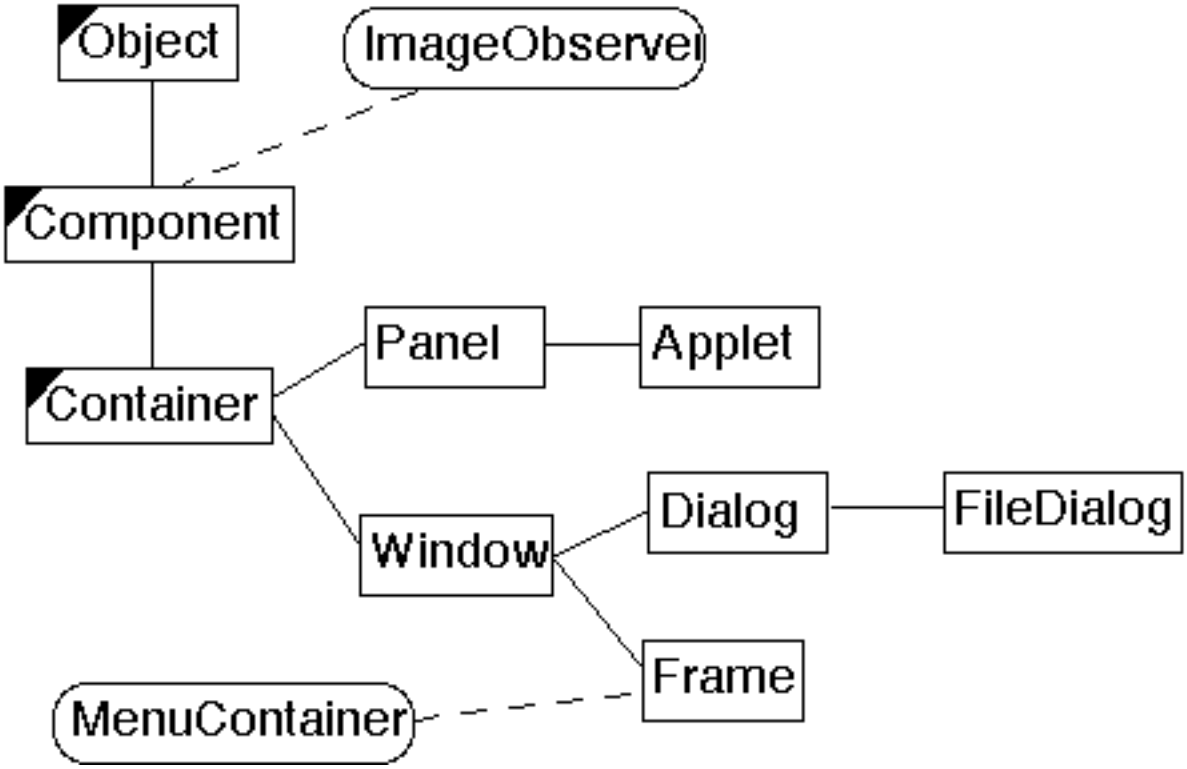
Use Composite pattern when you want

- To represent part-whole hierarchies of objects
- Clients to be able to ignore the difference between compositions of objects and individual objects

Java Use of Composite - AWT Widgets



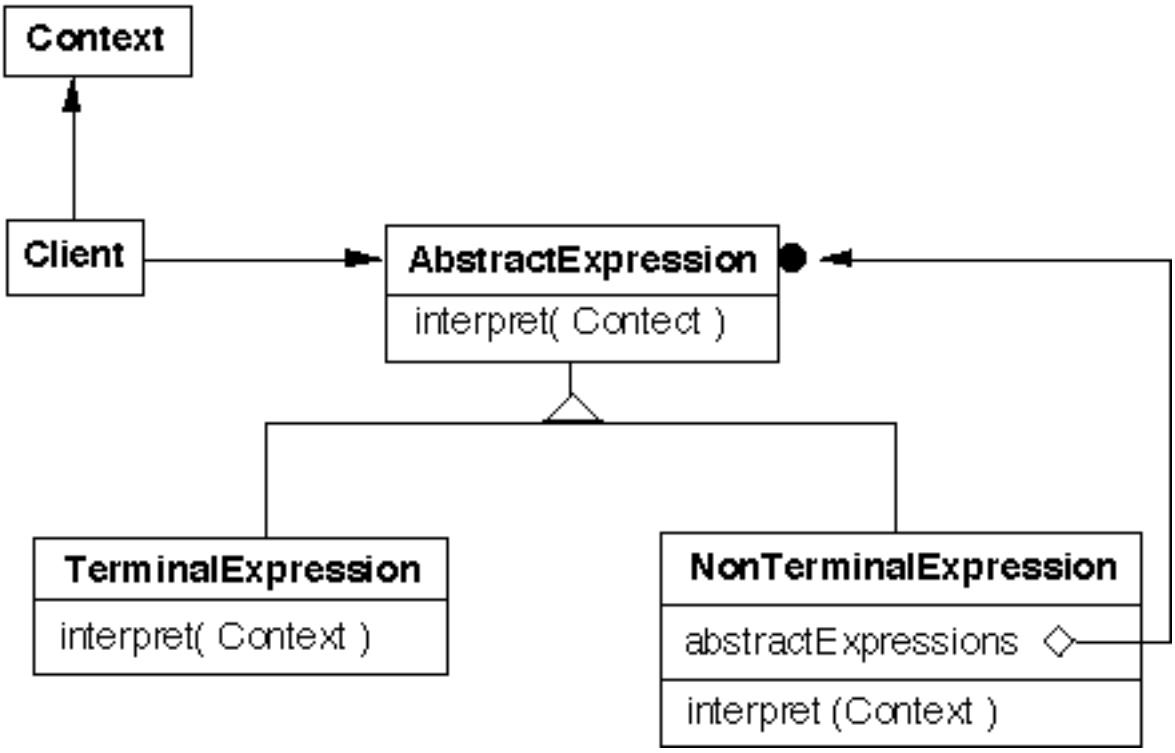
Specialized Java Containers



Interpreter

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language

Structure



Given a language defined by a simple grammar with rules like:

R ::= R1 R2 ... Rn

you create a class for each rule

The classes can be used to construct a tree that represents elements of the language

Example - Boolean Expressions

```
BooleanExpression ::=
            Variable        |
            Constant        |
            Or              |
            And             |
            Not             |
            BooleanExpression

And   ::= BooleanExpression 'and' BooleanExpression
Or    ::= BooleanExpression 'or' BooleanExpression
Not      ::= 'not' BooleanExpression

Constant ::= 'true' | 'false'
Variable ::= String

public interface BooleanExpression{
   public boolean evaluate( Context values );
   public BooleanExpression replace( String varName,
                        BooleanExpression replacement );
   public Object clone();
   public String toString();
}
```

And

And ::= BooleanExpression ‘&&’ BooleanExpression

```
public class And implements BooleanExpression {
   private BooleanExpression leftOperand;
   private BooleanExpression rightOperand;

   public And( BooleanExpression leftOperand,
               BooleanExpression rightOperand) {
      this.leftOperand = leftOperand;
      this.rightOperand = rightOperand;
   }

   public boolean evaluate( Context values ) {
      return leftOperand.evaluate( values ) &&
            rightOperand.evaluate( values );
   }

   public BooleanExpression replace( String varName,
         BooleanExpression replacement ) {
      return new And( leftOperand.replace( varName, replacement),
               rightOperand.replace( varName, replacement) );
   }

   public Object clone() {
      return new And( (BooleanExpression) leftOperand.clone( ),
               (BooleanExpression)rightOperand.clone( ) );
   }

   public String toString(){
      return "(" + leftOperand.toString() + " and " +
            rightOperand.toString() + ")";
   }
}
```

Or

Or ::= BooleanExpression ‘or’ BooleanExpression

```
public class Or implements BooleanExpression {
   private BooleanExpression leftOperand;
   private BooleanExpression rightOperand;

   public Or( BooleanExpression leftOperand,
              BooleanExpression rightOperand) {
      this.leftOperand = leftOperand;
      this.rightOperand = rightOperand;
   }

   public boolean evaluate( Context values ) {
      return leftOperand.evaluate( values ) ||
            rightOperand.evaluate( values );
   }

   public BooleanExpression replace( String varName,
         BooleanExpression replacement ) {
      return new Or( leftOperand.replace( varName, replacement),
              rightOperand.replace( varName, replacement) );
   }

   public Object clone() {
      return new Or( (BooleanExpression) leftOperand.clone( ),
              (BooleanExpression)rightOperand.clone( ) );
   }

   public String toString() {
      return "(" + leftOperand.toString() + " or " +
            rightOperand.toString() + ")";
   }
}
```

Not

Not ::= ‘not’ BooleanExpression

```
public class Not implements BooleanExpression {
   private BooleanExpression operand;

   public Not( BooleanExpression operand) {
      this.operand = operand;
   }

   public boolean evaluate( Context values ) {
      return  ! operand.evaluate( values );
   }

   public BooleanExpression replace( String varName,
       BooleanExpression replacement ) {
      return new Not( operand.replace( varName, replacement) );
   }

   public Object clone() {
      return new Not( (BooleanExpression) operand.clone( ) );
   }

   public String toString() {
      return "( not " +  operand.toString() + ")";
   }
```

Constant

Constant ::= ‘true’ | ‘false’

```
public class Constant implements BooleanExpression {
   private boolean value;
   private static Constant True = new Constant( true );
   private static Constant False = new Constant( false );

   public static Constant getTrue() {
      return True;
   }

   public static Constant getFalse(){
      return False;
   }

   private Constant( boolean value) {
      this.value = value;
   }

   public boolean evaluate( Context values ) {
      return  value;
   }

   public BooleanExpression replace( String varName,
         BooleanExpression replacement ) {
      return this;
   }

   public Object clone() {
      return this;
      }

   public String toString() {
      return String.valueOf( value );
   }
}
```

Variable

Variable ::= String

```
public class Variable implements BooleanExpression {
   private static Hashtable flyWeights = new Hashtable();

   private String name;

   public static Variable get( String name ) {
      if ( ! flyWeights.contains( name ))
         flyWeights.put( name , new Variable( name ));

      return (Variable) flyWeights.get( name );
   }

   private Variable( String name ) {
      this.name = name;
   }

   public boolean evaluate( Context values ) {
      return values.getValue( name );
   }

   public BooleanExpression replace( String varName,
         BooleanExpression replacement ) {
      if ( varName.equals( name ) )
         return (BooleanExpression) replacement.clone();
      else
         return this;
   }

   public Object clone() {
      return this;
   }

   public String toString() { return name; }
}
```

Context

```
public class Context {
   Hashtable values = new Hashtable();

   public boolean getValue( String variableName ) {
      Boolean wrappedValue = (Boolean) values.get( variableName );
      return wrappedValue.booleanValue();
   }

   public void setValue( String variableName, boolean value ) {
      values.put( variableName, new Boolean( value ) );
   }
}
```

Sample Use

```
public class Test {
   public static void main( String args[] ) throws Exception  {
      BooleanExpression left =
         new Or(  Constant.getTrue(), Variable.get( "x" ) );
      BooleanExpression right =
         new And(  Variable.get( "w" ), Variable.get( "x" ) );

      BooleanExpression all = new And(  left, right );

      System.out.println( all );
      Context values = new Context();
      values.setValue( "x", true );
      values.setValue( "w", false );

      System.out.println( all.evaluate( values ) );
      System.out.println( all.replace( "x", right ) );
      }
   }
```

Output

```
((true or x) and (w and x))
false
((true or (w and x)) and (w and (w and x)))
```

Consequences

It's easy to change and extend the grammar

Implementing the grammar is easy

Complex grammars are hard to maintain

Adding new ways to interpret expressions

The visitor pattern is useful here

Implementation

The pattern does not talk about parsing!

Flyweight

- If terminal symbols are repeated many times using the Flyweight pattern can reduce space usage
- The above example has each terminal class manage the flyweights for its objects, since Java does limited support for protecting constructors