CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2009
Doc 11 Composite, Interpreter, FlyWeight
March 3, 2009

Copyright ©, All rights reserved. 2009 SDSU & Roger Whitney, 5500 Campanile Drive, San Diego, CA 92182-7700 USA. OpenContent (http://www.opencontent.org/opl.shtml) license defines the copyright on this document.

References

Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides, 1995, pp. 243-272, 195-206

The Design Patterns Smalltalk Companion, Alpert, Brown, Woolf, 1998, pp. 261-273, 189-212

Reading

March 3 Proxy and State

March 5 Composite, Interpreter, Flyweight

March 10 Observer and Prototype

March 12

On the Criteria To Be Used in Decomposing Systems into Modules. Parnas, http://www.cs.umd.edu/class/spring2003/cmsc838p/Design/criteria.pdf

Object Coupling and Object Cohesion, chapter 7 of Essays on Object-Oriented Software Engineering, Vol. 1, Berard, Prentice-Hall, 1993, pp. 72-86

March 17

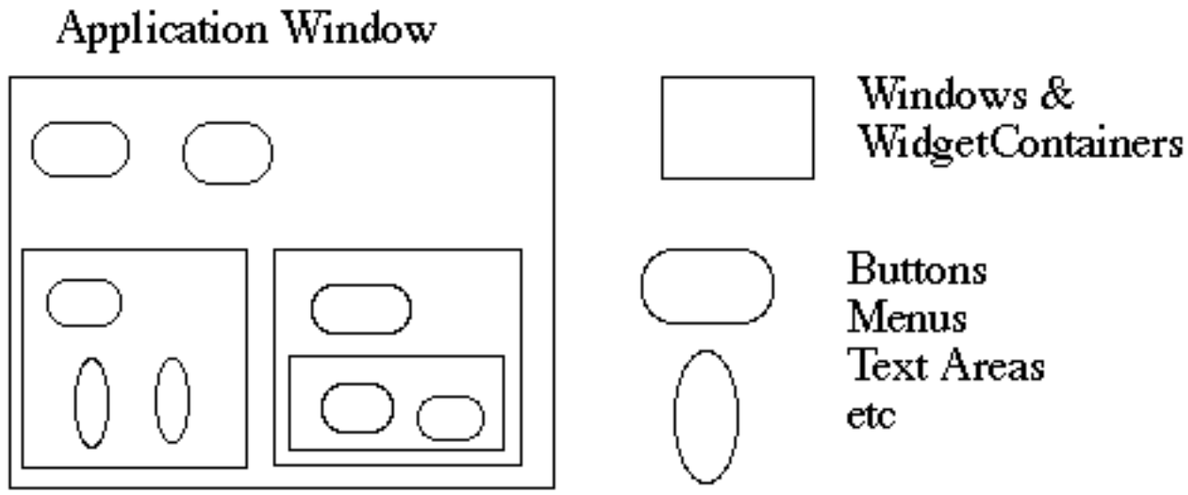
Object Coupling and Object Cohesion, chapter 7 of Essays on Object-Oriented Software Engineering, Vol. 1, Berard, Prentice-Hall, 1993, pp 92-111

March 19

Object Coupling and Object Cohesion, chapter 7 of Essays on Object-Oriented Software Engineering, Vol. 1, Berard, Prentice-Hall, 1993, pp 86-92, 111-122

Talk about Exam and Assignment 3 dates

Composite Motivation



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

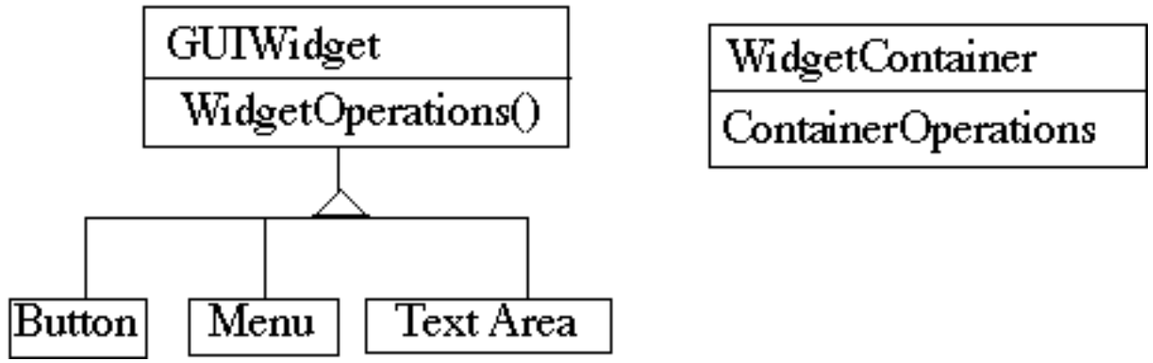
Widgets are different that WidgetContainers

Bad News

```
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 (myButtons != null)
        etc.
```

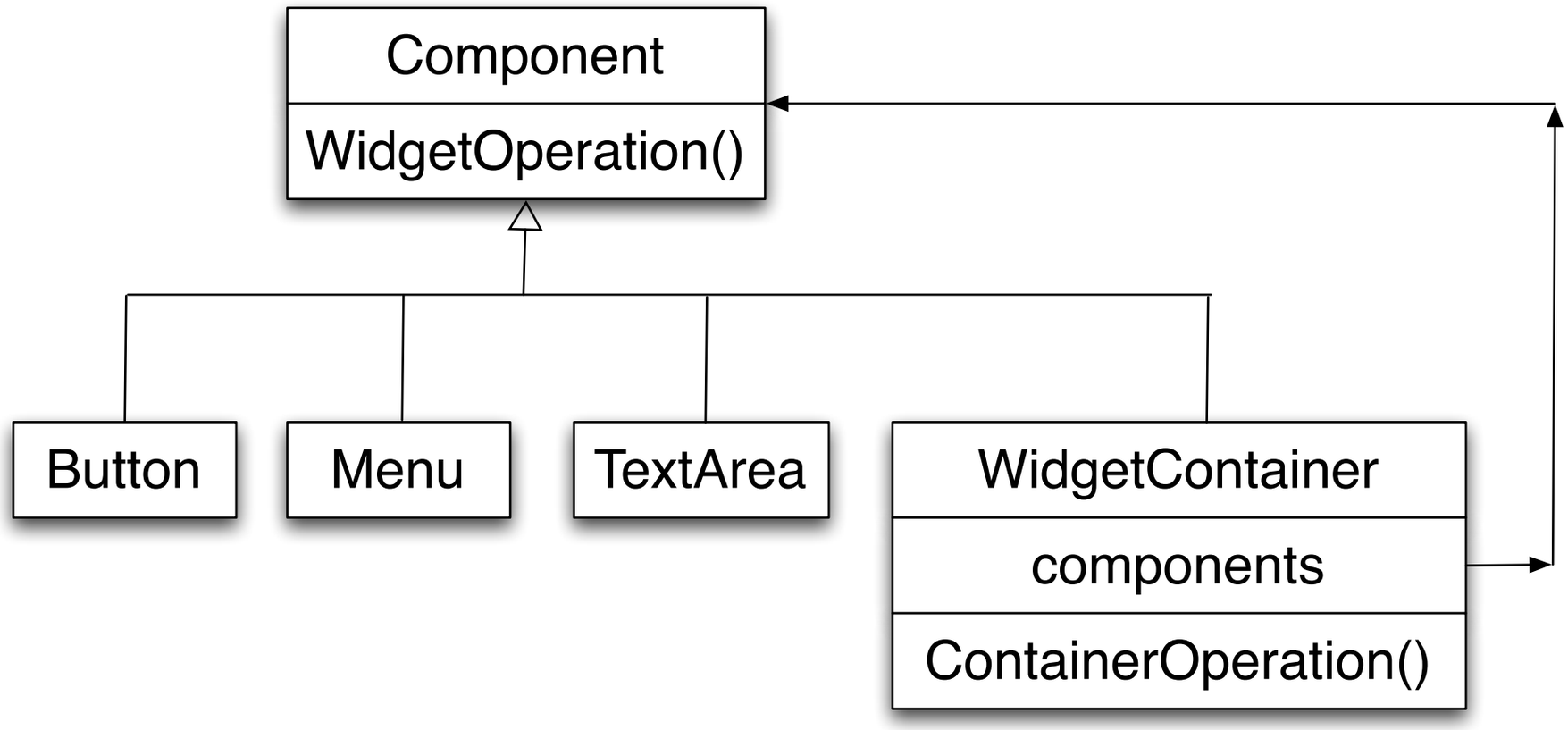
An Improvement




```
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.
        }
}
```

Composite Pattern



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 Opeartions

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

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

Issue - Parent References

```
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?

Applicability

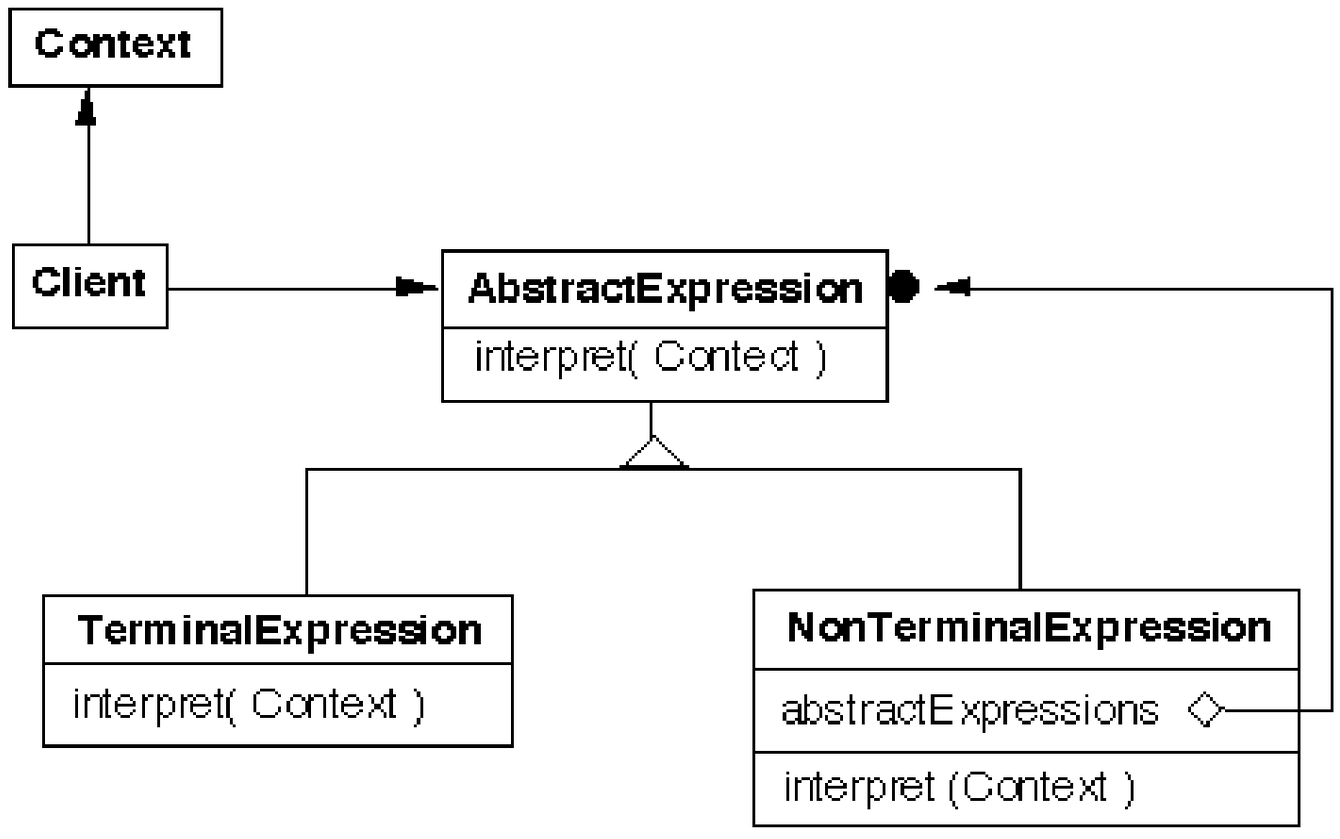
Use Composite pattern when you want

To represent part-whole hierarchies of objects

Clients to be able to ignore the difference between compositions of objects and individual objects

Interpreter

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



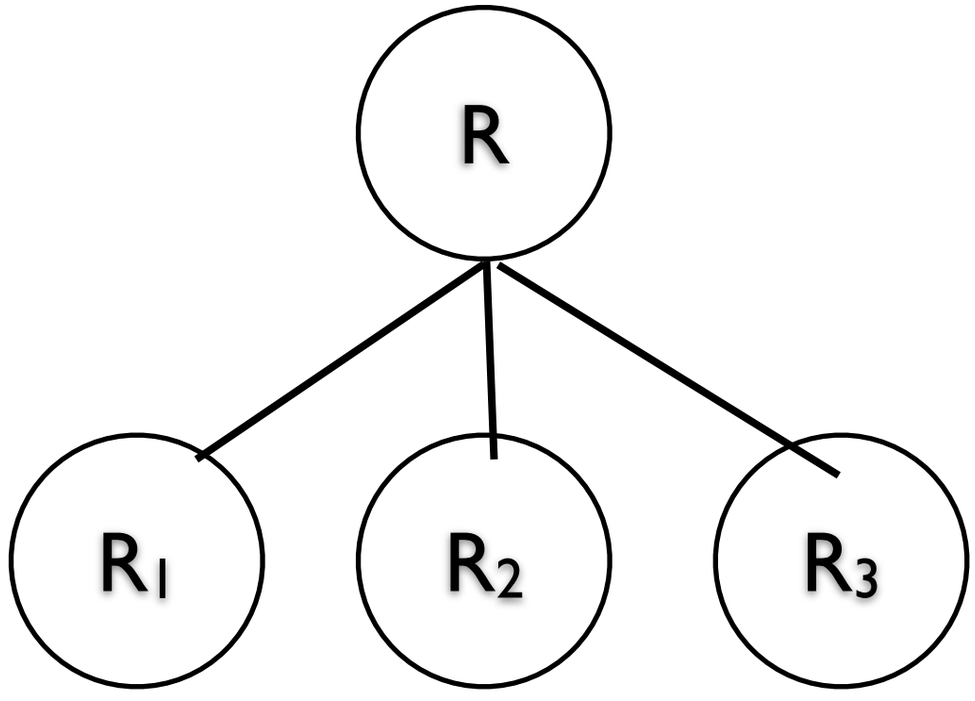
Grammar & Classes

Given a language defined by a grammar like:

$$R ::= R_1\ R_2\ R_3$$

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 ::=

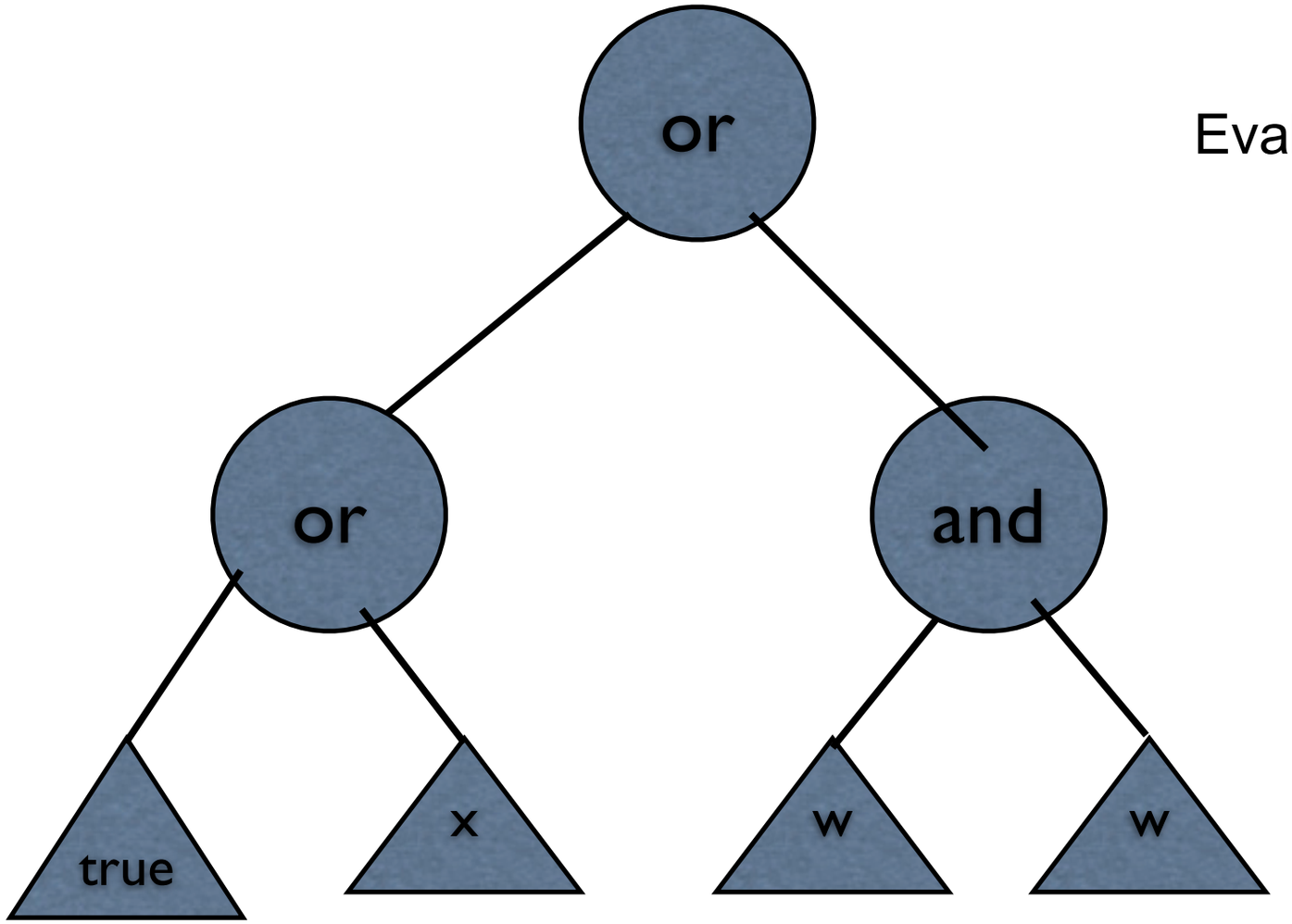
Variable	|
Constant	|
Or	|
And	|
Not	|
BooleanExpression	

And ::= '(' BooleanExpression ‘and’ BooleanExpression ')'
Or ::= '(' BooleanExpression ‘or’ BooleanExpression ')'
Not ::= ‘not’ BooleanExpression

Constant ::= ‘true’ | ‘false’
Variable ::= String

Sample Expression

((true or x) or (w and x))



Evaluate with

x = true

w = false

Sample Classes

```
public interface BooleanExpression{
        public boolean evaluate( Context values );
        public String toString();
}
```

And

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

Constant

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

Variable

```
public class Variable implements BooleanExpression {

        private String name;

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

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

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

Context

```
public class Context {
	Hashtable<String,Boolean> values = new Hashtable<String,Boolean>();

	public boolean getValue( String variableName ) {
		return values.get( variableName );
	}

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

((true or x) or (w and x))

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

        BooleanExpression all = new Or(  left, right );

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

        System.out.println( all.evaluate( values ) );
        }
}
```

Consequences

It's easy to change and extend the grammar

Implementing the grammar is easy

Complex grammars are hard to maintain

- Use JavaCC or SmaCC instead

Adding new ways to interpret expressions

- The visitor pattern is useful here

Complicates design when a language is simple

Supports combinations of elements better than implicit language

Implementation

The pattern does not talk about parsing!

Flyweight

If terminal symbols are repeated many times using the Flyweight pattern can reduce space usage

Composite

Abstract syntax tree is an instance of the composite

Iterator

Can be used to traverse the structure

Visitor

Can be used to place behavior in one class

Flyweight

Use sharing to support large number of fine-grained objects efficiently

Text Example

A document has many instances of the character 'a'

Character has

- Font
- width
- Height
- Ascenders
- Descenders
- Where it is in the document

Most of these are the same for all instances of 'a'

Use one object to represent all instances of 'a'

Java String Example

```
public void testInterned() {
        String a1 = "catrat";
        String a2 = "cat";
        assertFalse(a1 == (a2+ "rat"));

        String a3 = (a2 + "rat").intern();
        assertTrue(a1 == a3);
        String a4 = "cat" + "rat";
        assertTrue(a1 == a4);
        assertTrue(a3 == a4);
    }
```

public String intern()

Returns a canonical representation for the string object.

A pool of strings, initially empty, is maintained privately by the class String.

Intrinsic State

Information that is independent from the objects context

The information that can be shared among many objects

So can be stored outside of the flyweight

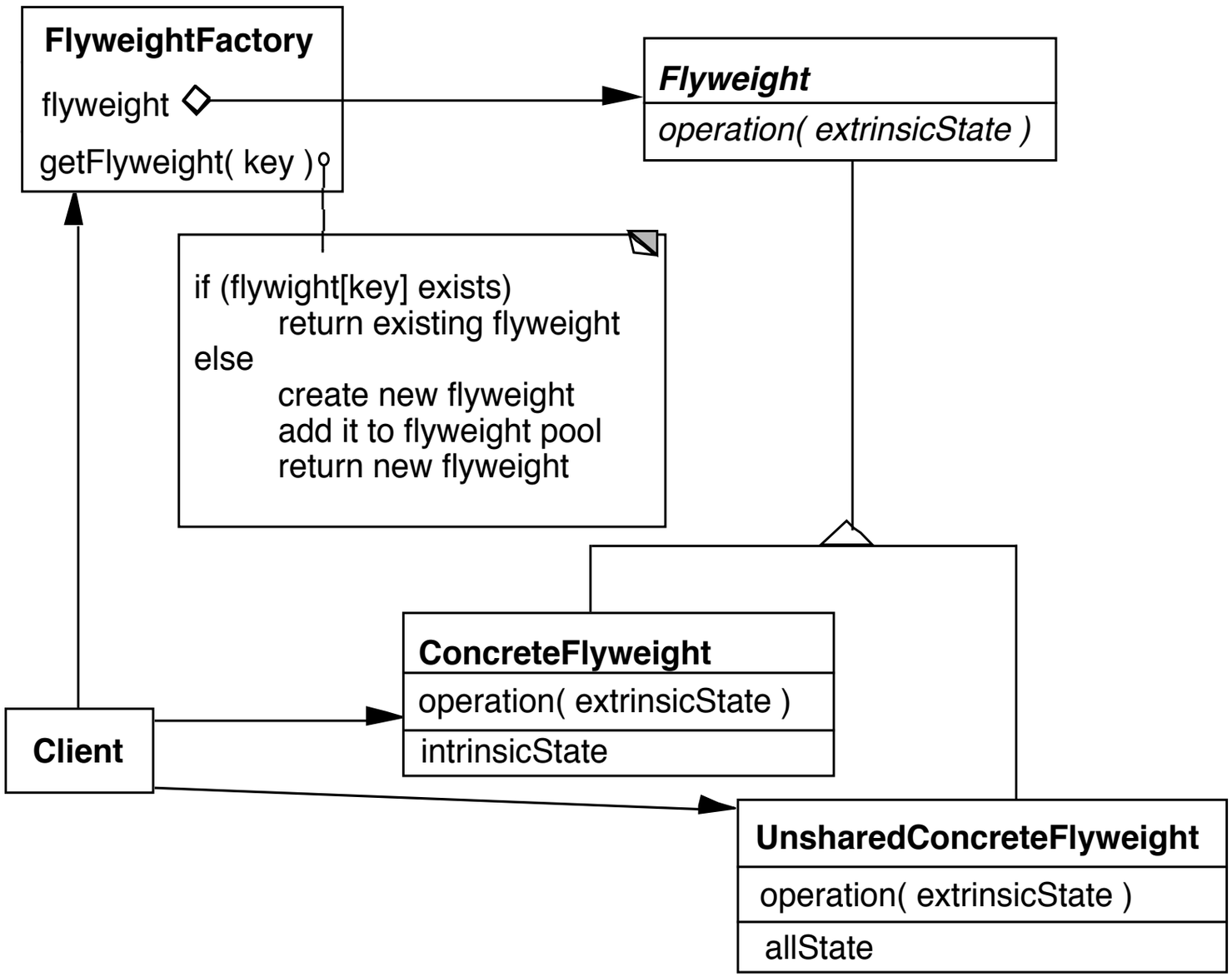
Extrinsic State

Information that is dependent on the objects context

The information that can not be shared among objects

So has to be stored outside of the flyweight

Structure


FlyweightFactory
flyweight
getFlyweight(key)
Flyweight
operation(extrinsicState)
if (flywight[key] exists)
return existing flyweight
else
create new flyweight
add it to flyweight pool
return new flyweight
ConcreteFlyweight
operation(extrinsicState)
intrinsicState
Client
UnsharedConcreteFlyweight
operation(extrinsicState)
allState

The Hard Part

Separating state from the flyweight

Where does the extrinsic state go?

How do we get extrinsic state to correct object when it is needed?