CS 635 Advanced Object-Oriented Design & Programming
Fall Semester, 2022
Doc 12 Python Decorator, Singleton
Oct 6, 2022

Singleton

Warning

Simplest pattern

- But has subtle issues particularly in Java

Most controversial pattern

Intent

Ensure a class only has one instance

Provide global point of access to single instance

Singleton

```
public class Counter {
    private int count = 0;
    private static Counter instance;
    private Counter() { }

    public static Counter instance() {
        if (instance == null)
            instance = new Counter();
        return instance;
    }

    public int increase()  {return ++count;}
}
```

One instance

Global access

Some Uses

Java Security Manager

Logging a Server

Null Object

Globals are Evil

Why Singletons Are Controversial(Evil)

Singletons provide global access point for some service

Hidden dependencies

Is there a different design that does not need singletons

Pass a reference

Why Singletons Are Controversial(Evil)

Singletons allow you to limit creation of objects of a class

Should that be the responsibility of the class?

Class should do one thing

Use factory or builder to limit the creation

Why Singletons Are Controversial(Evil)

Singletons tightly couple you to the exact type of the singleton object

No polymorphism

Hard to subclass

Why Singletons Are Controversial(Evil)

Singletons carry state with them that last as long as the program lasts

Persistent state makes testing hard and error prone

Why Singletons Are Controversial(Evil)

A Singleton today is a multiple tomorrow

Singleton pattern makes it hard to change to allow multiple objects

Why Singletons Are Controversial(Evil)

In Java Singletons are a lie

More on this later

Singleton Implementation - Python

Strait forward Translation

```
class Foo:
    _instance = None

    def __init__(self):
        raise RuntimeError('Call instance() instead')

    @classmethod
    def instance(cls):
        if cls._instance is None:
            print('Creating new instance')
            cls._instance = cls.__new__(cls)

        return cls._instance
```

```
x = Foo.instance()
```

https://python-patterns.guide/gang-of-four/singleton/

Simpler Implementation

```
class Foo(object):
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            print('Creating the object')
            cls._instance = super(Logger, cls).__new__(cls)
            # Put any initialization here.
        return cls._instance


x = Foo()
```

Singleton Implementation - Java

Why Not Use This?

```
public class Counter {
    private static int count = 0;

    public static int increase()  {return ++count;}
}
```

Why Not Use This?

```
public class Counter {
    private int count = 0;
    private Counter() { }

    public static Counter instance = new Counter();

    public int increase()  {return ++count;}
}
```

Two Useful Features

Lazy

Only created when needed

Thread safe

Recommended Implementation

```
public class Counter {
    private int count = 0;
    private Counter() { }

    private static class SingletonHolder {
       private final static Counter INSTANCE = new Counter();
    }

    public static Counter instance() {
          return SingletonHolder.INSTANCE;
    }

    public int increase()  {return ++count;}
}
```

Correct but not Lazy

```
public class Counter {
    private int count = 0;
    protected Counter() { }

    private final static Counter INSTANCE = new Counter();

    public static Counter instance() {
            return INSTANCE;
    }

    public int increase()  {return ++count;}
}
```

Lazy, Thread safe with Overhead

```
public class Counter {
    private int count = 0;
    private static Counter instance;
    private Counter() { }

    public static synchronized Counter instance() {
        if (instance == null)
            instance = new Counter();
        return instance();
    }

    public int increase()  {return ++count;}
}
```

Java Templates & Singleton

Does not compile

```
public class TemplateSingleton<Type> {
    Type foo;

    public static TemplateSingleton<Type> instance =
        new TemplateSingleton<Type>();
}
```

When is a Singleton not a Singleton?

When Java Garbage Collects Classes

Singleton class can be garbage collected
Singleton loses any value it had

Solution

Turn off garbage collection of classes (-Xnoclassgc)

Make sure there is always a reference to the class/instance

When Multiple Java Class Loaders are Used

When loaded by two different class loaders there will be two versions of the class

Some servlet engines use different class loader for each servlet

Using custom class loaders can cause this

Purposely Reloading a Java Class

Servlet engines can force a class to be reloaded

Serialize and Deserialize Singleton Object

Serialize the singleton

Deserialize the singleton

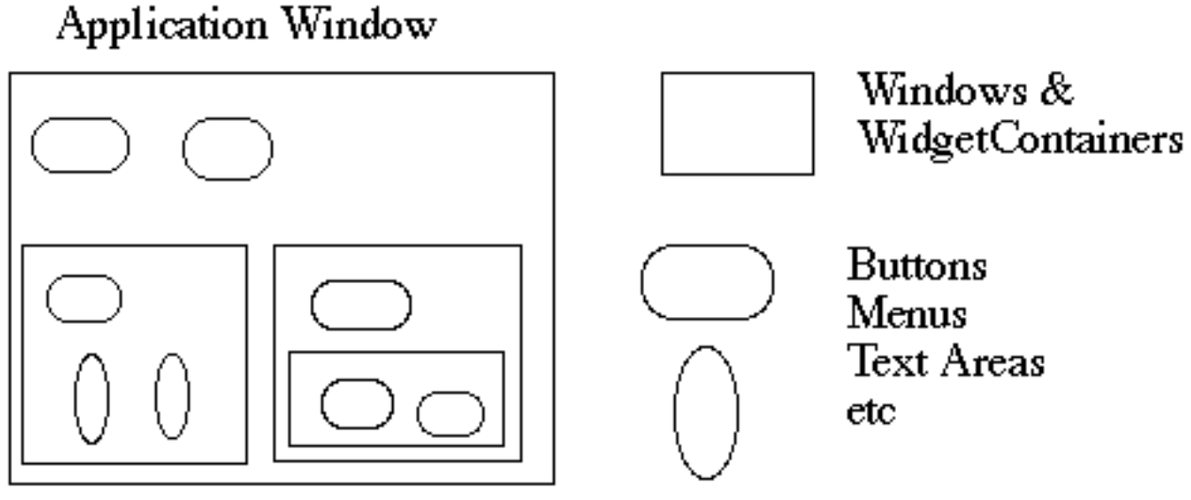
You now have two copies

One way to serialize a Java object is using ObjectOutputStream

Ruby Marshal.dump() will not marshal a singleton

Composite

Composite Motivation



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

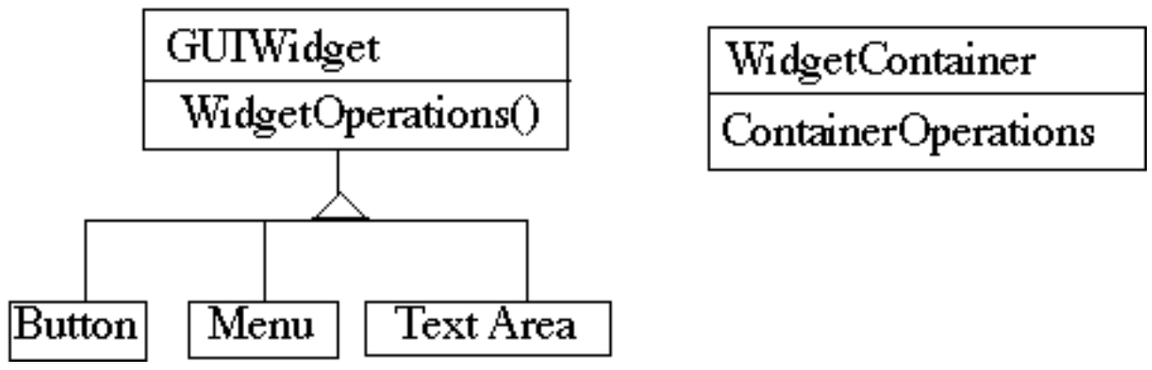
Widgets are different that WidgetContainers

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.
```

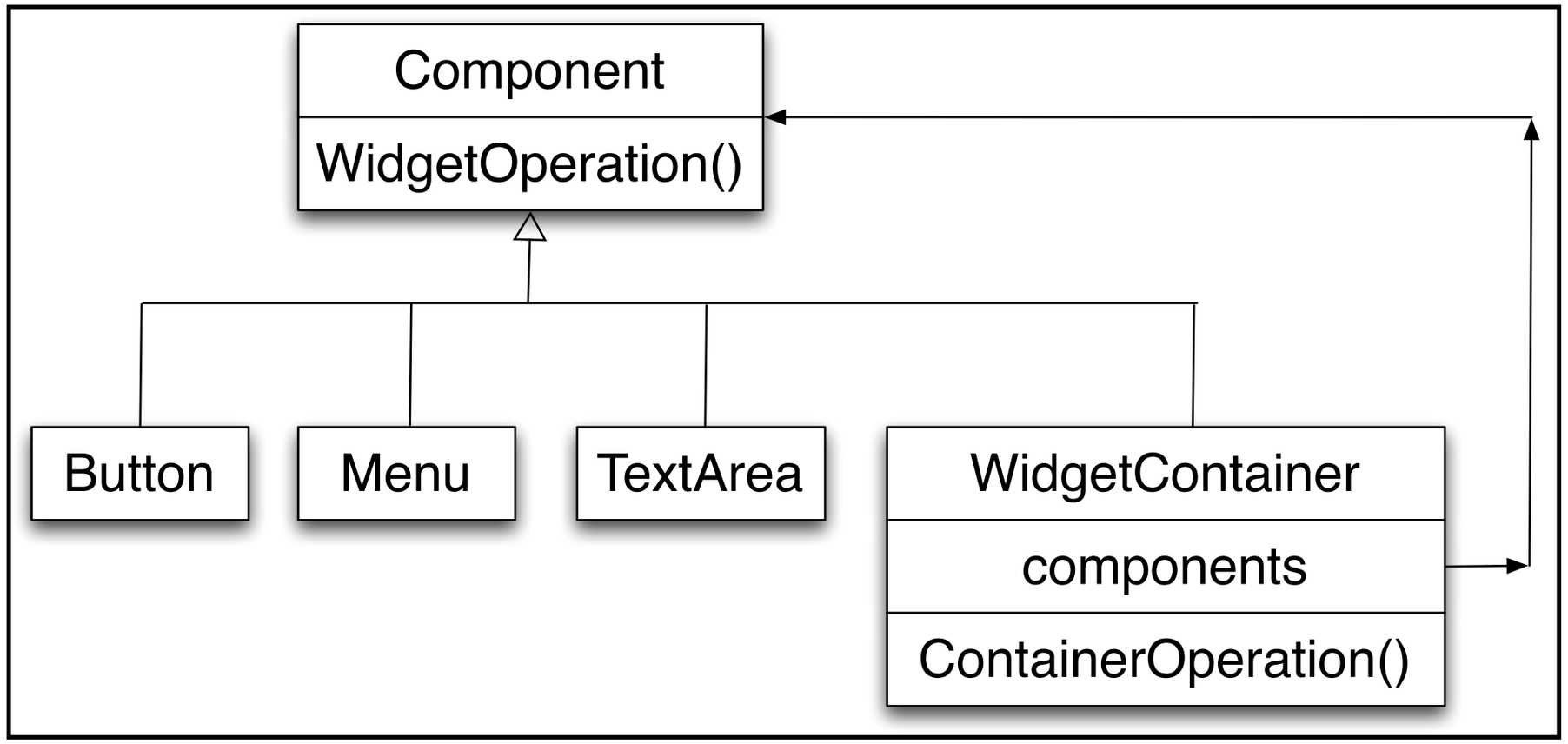
An Improvement




```
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



Intent

Compose objects into tree structures to represent part-whole hierarchies.

Lets clients treat individual objects and compositions of objects uniformly

Conflict

Lets clients treat individual objects and compositions of objects uniformly

Composites have add/remove operations that individual objects do not

Composite Pattern

Component implements default behavior for widgets when possible

Button, Menu, etc overrides Component methods when needed

WidgetContainer will have to override 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

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

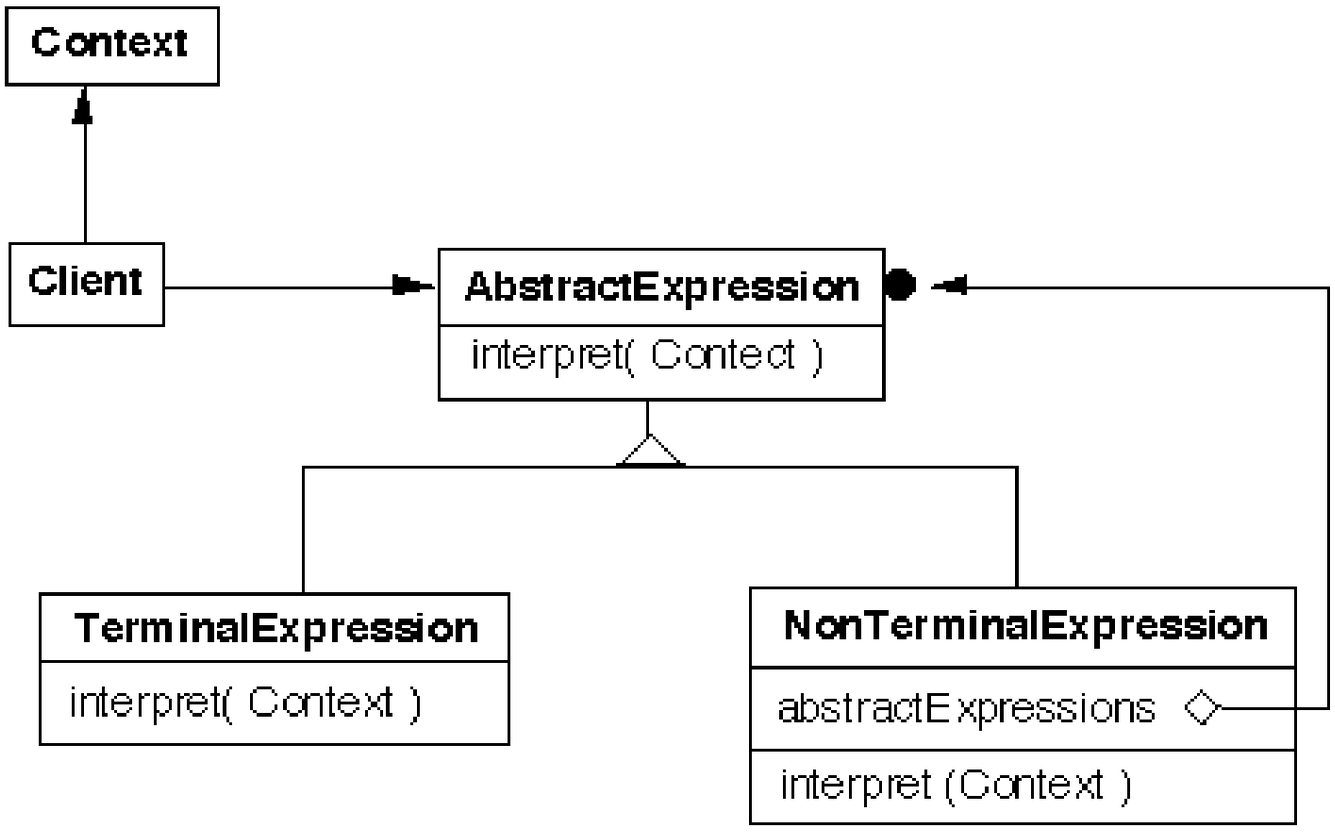
To represent part-whole hierarchies of objects

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

Interpreter

Interpreter

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



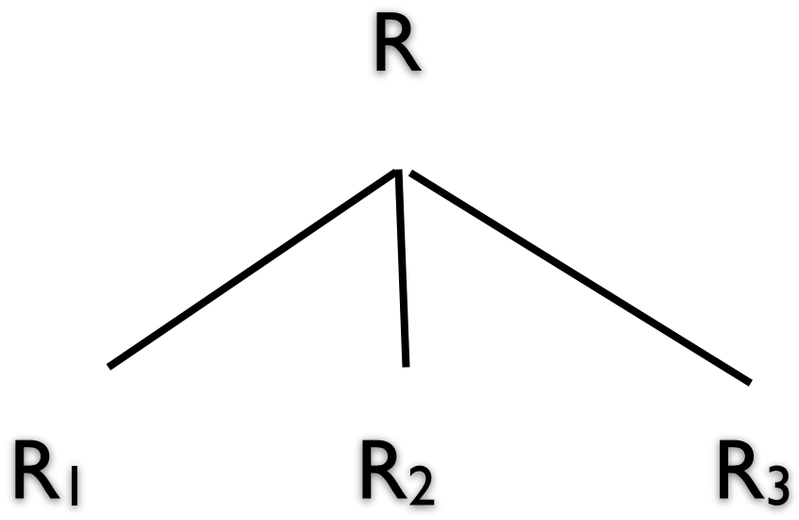
Grammar & Classes

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 ')'

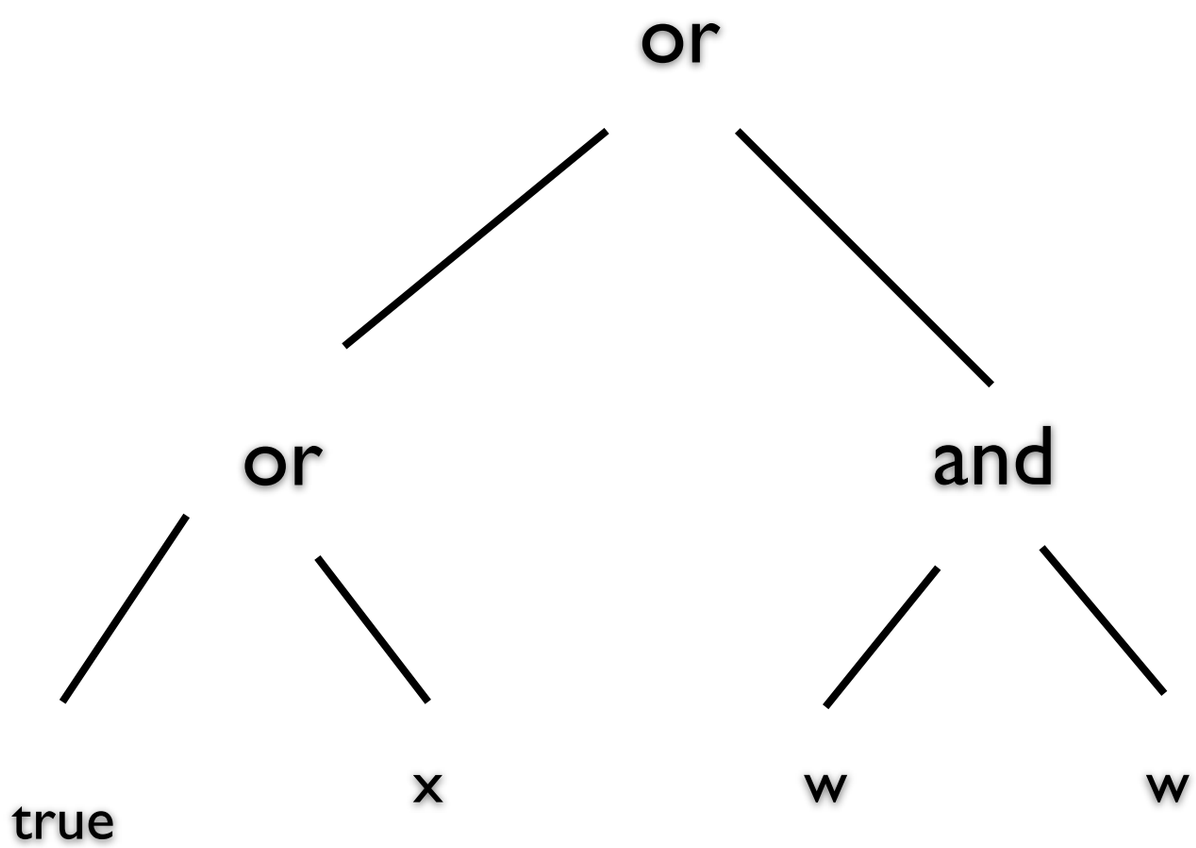
Not ::= ‘not’ BooleanExpression

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