

CS 635 Advanced Object-Oriented Design & Programming  
Spring Semester, 2011  
Doc 15 Adapter, Bridge, Interpreter  
April 12, 2011

Copyright ©, All rights reserved. 2011 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. 139-150, 151-162, 243-256

The Design Patterns Smalltalk Companion, Alpert, Brown, Woolf, 1998, 121-136

# Adapter



# Adapter

Convert interface of a class into another interface

Use adapter when

- You want to use an existing class but does not have interface on needs

- You want to create a reusable class that works with unrelated or unforeseen classes

# Address Book & JTable

Display an AddressBook object in a JTable

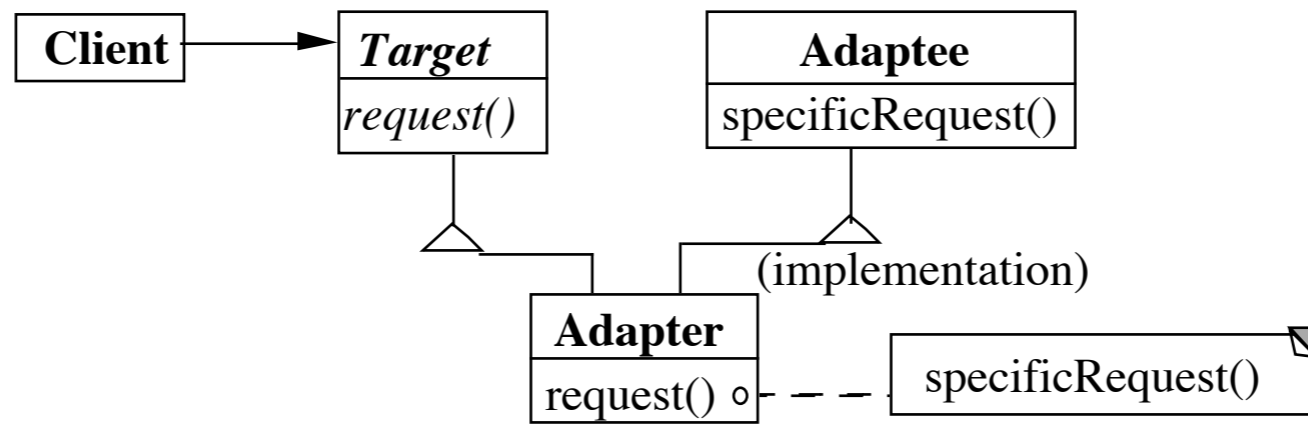
JTables require objects of type TableModel

```
public class AddressBook{
    List personList;
    public int getSize(){...}
    public int addPerson(...){...}
    public Person getPerson(...){...}
    ...
}
```

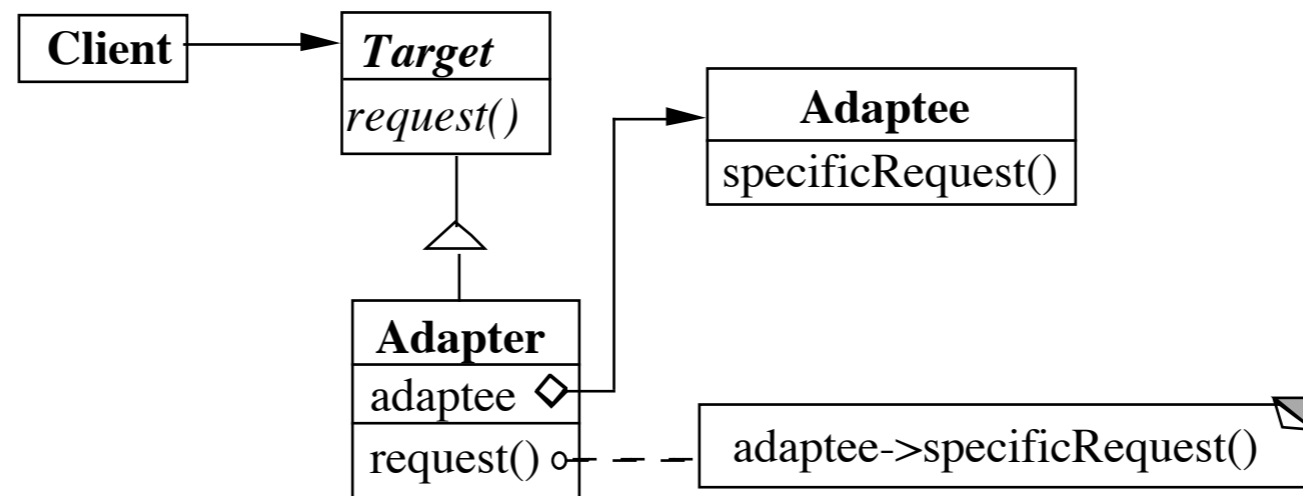
```
public class AddressBookTableAdapter implements TableModel
{
    AddressBook ab;
    public AddressBookTableAdapter( AddressBook ab ){
        this.ab = ab;
    }
    //TableModel impl
    public getRowCount(){
        ab.getSize();

    public Object getValueAt(int rowIndex, int columnIndex) {
        Person requested =
            ad.getPerson(convertRowToName(rowIndex));
        return requested.get(convert(columnIndex));
    }
}
```

# Class Adapter



# Object Adapter



# Class Adapter Example

```
class OldSquarePeg {  
    public: void squarePegOperation() { do something }  
}
```

```
class RoundPeg {  
    public: void virtual roundPegOperation = 0;  
}
```

```
class PegAdapter: private OldSquarePeg, public RoundPeg {  
    public:  
        void virtual roundPegOperation() {  
            add some corners;  
            squarePegOperation();  
        }  
}
```

```
void clientMethod() {  
    RoundPeg* aPeg = new PegAdapter();  
    aPeg->roundPegOperation();  
}
```

# Object Adapter

```
class OldSquarePeg{
    public: void squarePegOperation() { do something }
}

class RoundPeg    {
    public: void virtual roundPegOperation = 0;
}

class PegAdapter: public RoundPeg    {
    private:
        OldSquarePeg* square;

    public:
        PegAdapter() { square = new OldSquarePeg; }

        void virtual roundPegOperation()    {
            add some corners;
            square->squarePegOperation();
        }
}
```



How Much Adapting does the Adapter do?

# Two-way Adapters

```
class OldSquarePeg {
    public:
        void virtual squarePegOperation() { blah }
}

class RoundPeg {
    public:
        void virtual roundPegOperation() { blah }
}

class PegAdapter: public OldSquarePeg, RoundPeg {
    public:
        void virtual roundPegOperation() {
            add some corners;
            squarePegOperation();
        }
        void virtual squarePegOperation() {
            add some corners;
            roundPegOperation();
        }
}
```

# Flasher and MouseListener

```
class Flasher
  def toggle()
    @flashing = !@flashing
  end

  def pause()
    #etc
  end

  def resume()
    #etc
  end
end
```

```
class MouseListener
  def mouseClicked(event)
  end

  def mouseEntered(event)
  end

  def mouseExited(event)
  end
end
```

## Actions we want

mouse click toggles flasher  
mouse enter pauses  
mouse exits resumes

# Flasher as MouseListener

```
class Flasher
  def toggle()
    @flashing = !@flashing
  end

  def pause()
    #etc
  end

  def resume()
    #etc
  end

  def mouseClicked(event)
    toggle()
  end

  def mouseEntered(event)
    pause()
  end

  def mouseExited(event)
    resume()
  end
end
```

# Simple Adapter

```
class Flasher
  def toggle()
    @flashing = !@flashing
  end

  def pause()
    #etc
  end

  def resume()
    #etc
  end
end

yellowFlasher = Flasher.new(yellow, fast)
FlasherAdapter.new(yellowFlasher)
```

```
class FlasherAdaptor
  def initialize(aFlasher)
    @flasher = aFlasher
  end

  def mouseClicked(event)
    @flasher.toggle()
  end

  def mouseEntered(event)
    @flasher.pause()
  end

  def mouseExited(event)
    @flasher.resume()
  end
end
```

# A Ruby Adapter - Forwardable

```
class Flasher
  def toggle()
    @flashing = !@flashing
  end

  def pause()
    #etc
  end

  def resume()
    #etc
  end
end
```

```
require 'forwardable'

class FlasherMouseListener
  extend Forwardable

  def initialize()
    @flasher = Flasher.new()
  end

  def _delegator(:@flasher, :toggle, :mouseClick)
  def _delegator(:@flasher, :pause, :mouseenter)
  def _delegator(:@flasher, :resume, :mouseExit)

end
```

```
adaptor = FlasherMouseListener.new()
adaptor.mouseClick()
```

# Parameterized Adapter

```
class MouseListenerAdapter
  def initialize(adaptee, clickMethod, enterMethod, exitMethod)
    @adaptee = adaptee
    @clickMethod = clickMethod
    @enterMethod = enterMethod
    @exitMethod = exitMethod
  end

  def mouseClicked(event)
    @adaptee.send(clickMethod)
  end

  def mouseEntered(event)
    @adaptee.send(clickMethod)
  end

  def mouseExited(event)
    @adaptee.send(clickMethod)
  end
end
```

```
yellowFlasher = Flasher.new(yellow, fast)
MouseListenerAdapter.new(
  yellowFlasher,
  :toggle,
  :pause,
  :resume)
```

# Better Parameterized Adapter

```
class MouseListenerAdapter
```

```
  def initialize(adaptee, clickLambda, enterLambda, exitLambda)
```

```
    @adaptee = adaptee
```

```
    @clickLambda = clickLambda
```

```
    @enterLambda = enterLambda
```

```
    @exitLambda = exitLambda
```

```
  end
```

```
  def mouseClicked(event)
```

```
    @clickLambda.call(adaptee)
```

```
  end
```

```
  def mouseEntered(event)
```

```
    @enterLambda.call(adaptee)
```

```
  end
```

```
  def mouseExited(event)
```

```
    @exitLambda.call(adaptee)
```

```
  end
```

```
end
```

```
yellowFlasher = Flasher.new(yellow, fast)
```

```
MouseListenerAdapter.new(
```

```
  yellowFlasher,
```

```
  lambda {|flasher| flasher.toggle()},
```

```
  lambda {|flasher| flasher.pause()},
```

```
  lambda {|flasher| flasher.resume()})
```



# What is this lambda?

no name function that remembers its environment

```
a = lambda {|param| puts(param)}
```

```
a.call(4)          #4
```

```
b = 5
```

```
c = lambda {|param| puts(param + b)}
```

```
c.call(4)          #9
```

```
def hideB(aLambda)
```

```
  b = 10
```

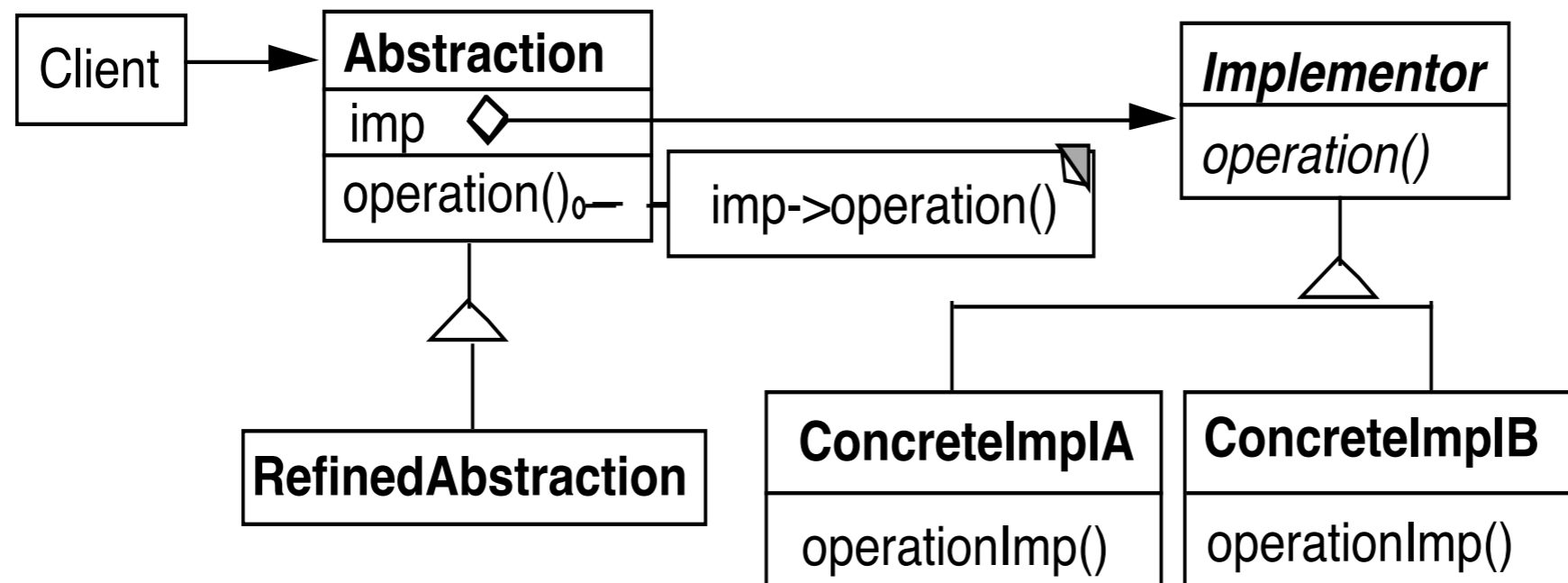
```
  aLambda.call(4)
```

```
end
```

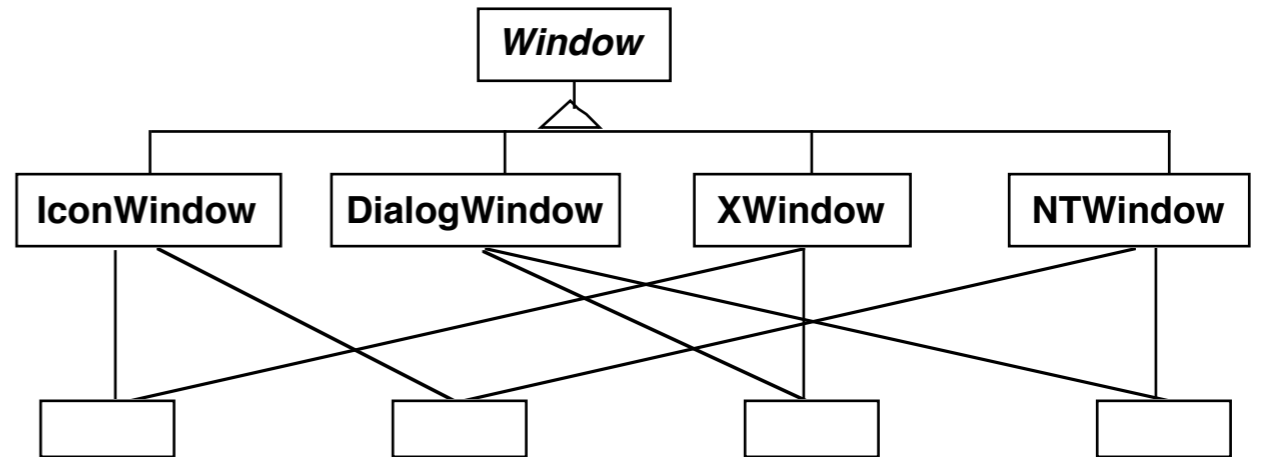
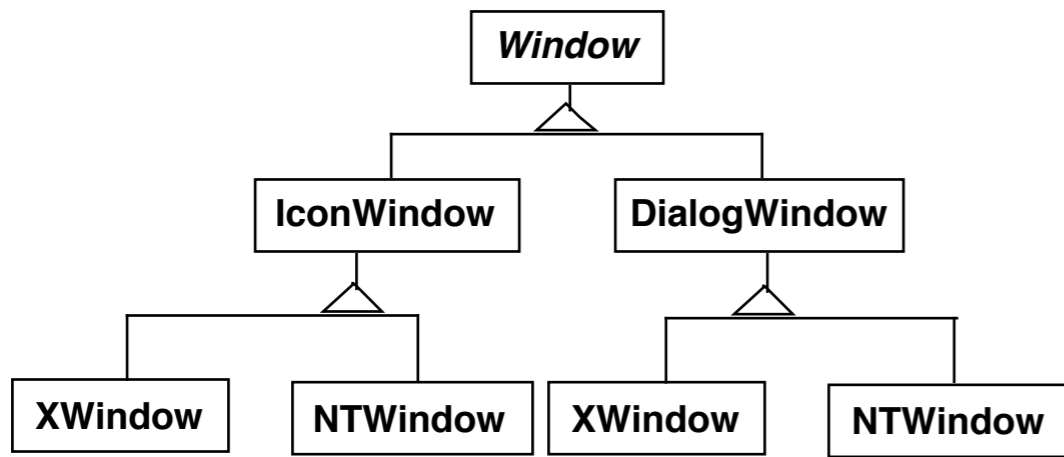
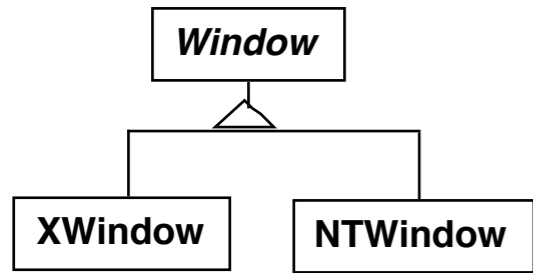
```
hideB(c)           #9
```

# Bridge

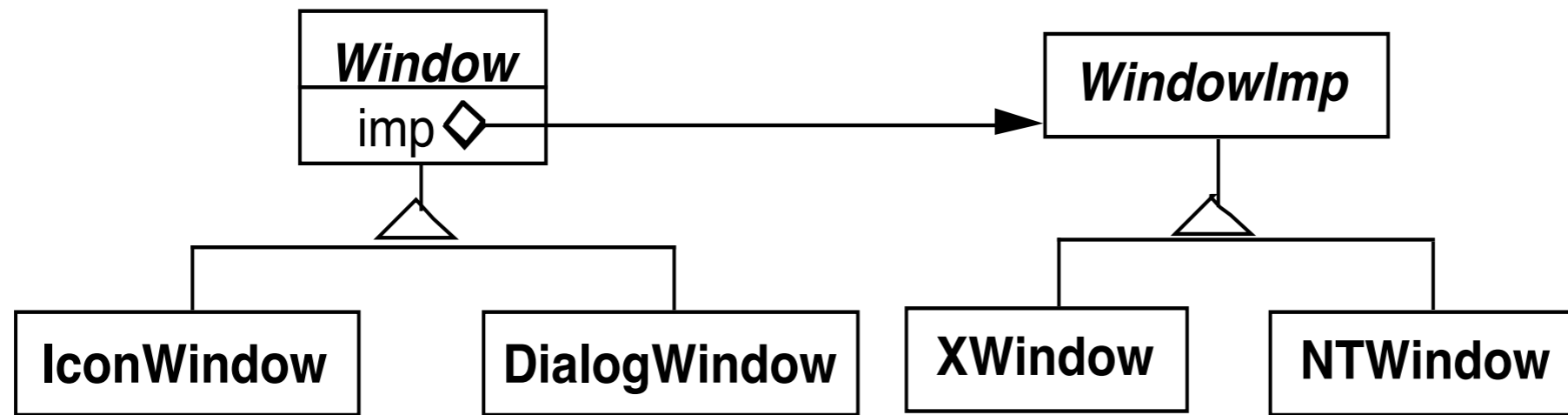
Decouple an abstraction from its implementation



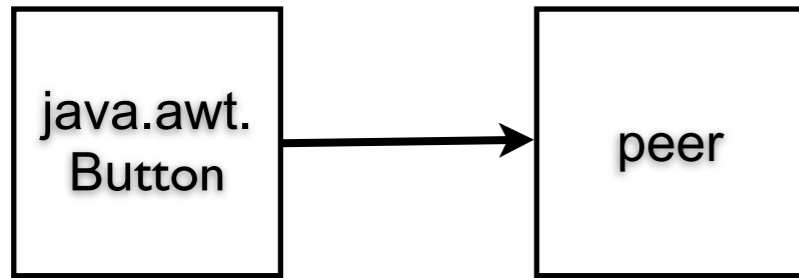
# Windows



# Using the Bridge Pattern



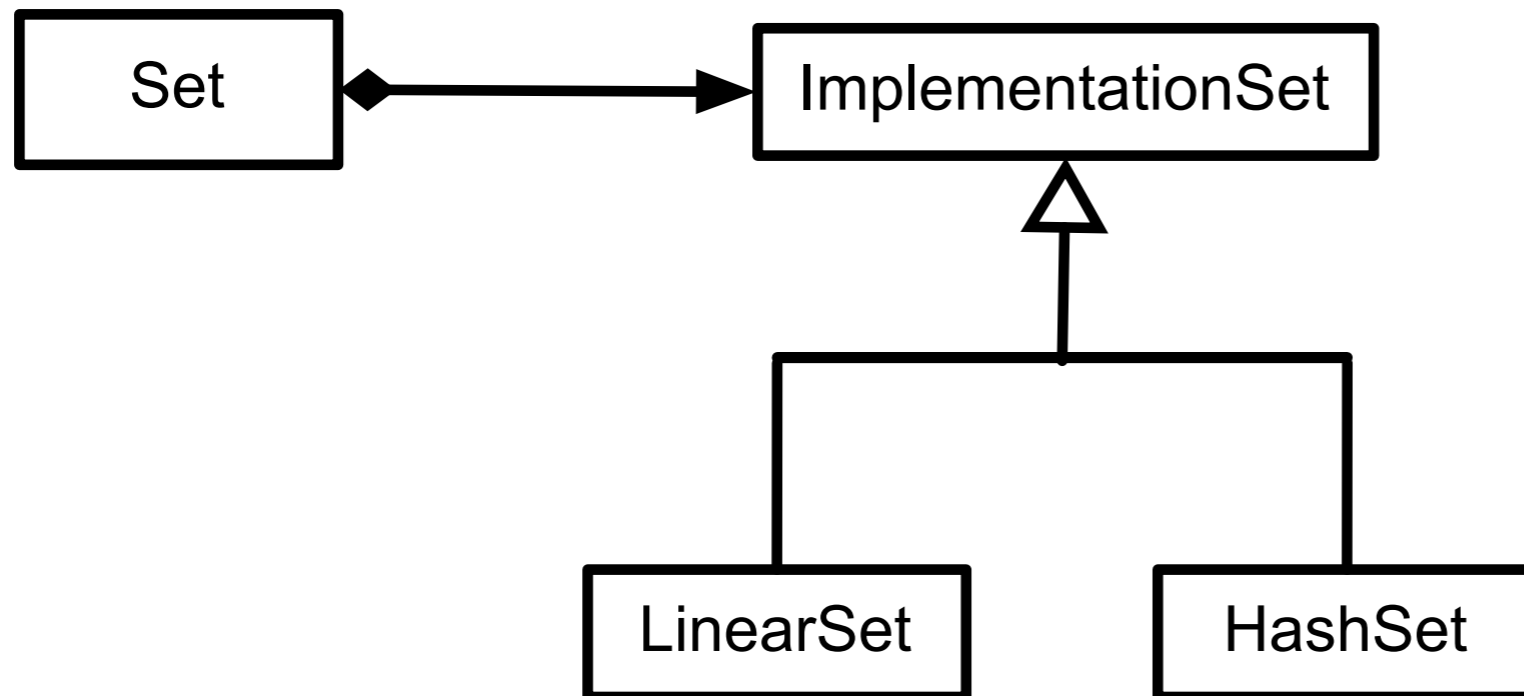
# Peers in Java's AWT



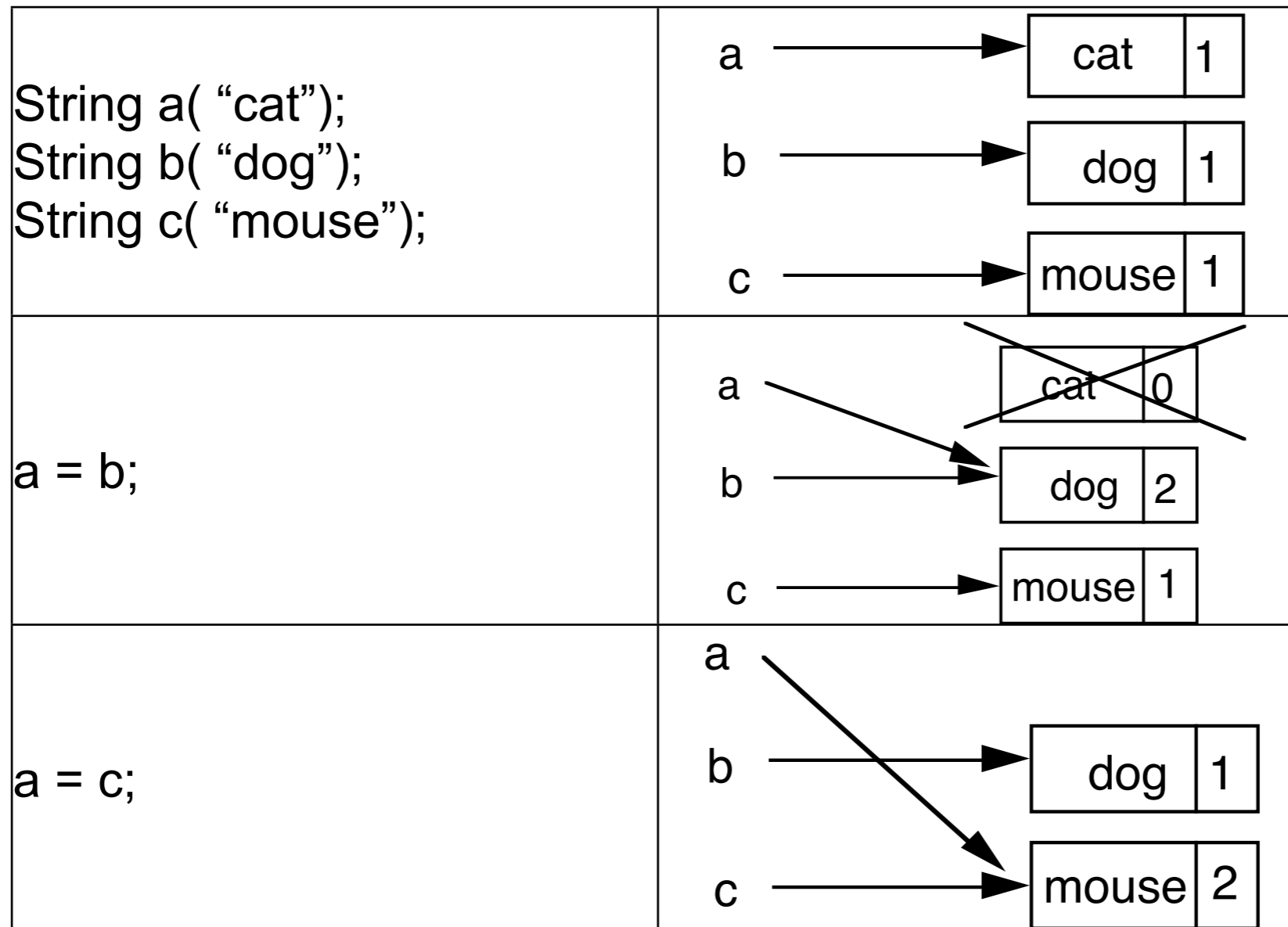
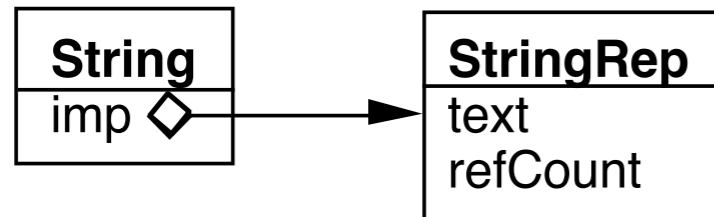
Peer = implementation

```
public synchronized void setCursor(Cursor cursor) {  
    this.cursor = cursor;  
    ComponentPeer peer = this.peer;  
    if (peer != null) {  
        peer.setCursor(cursor);  
    }  
}
```

# IBM Smalltalk Collections



# Smart Pointers in C++



# Coplien's Implementation

```
class StringRep {
    friend String;

private:
    char *text;
    int refCount;

    StringRep()      { *(text = new char[1] = '\0'); }

    StringRep( const StringRep& s ) {
        ::strcpy( text = new char[::strlen(s.text) + 1, s.text);
    }

    StringRep( const char *s)      {
        ::strcpy( text = new char[::strlen(s) + 1, s);
    }

    StringRep( char** const *r)    {
        text = *r;
        *r = 0;
        refCount = 1;;
    }

    ~StringRep()    { delete[] text; }
    int length() const { return ::strlen( text ); }
    void print() const { ::printf("%s\n", text ); }
}
}
```



```

class String    {
    friend StringRep
    public:
        String operator+(const String& add) const { return *imp + add; }
        StringRep* operator->() const      { return imp; }
        String()    { (imp = new StringRep()) -> refCount = 1;    }
        String(const char* charStr)  { (imp = new StringRep(charStr)) -> refCount = 1; }
        String operator=( const String& q) {
            (imp->refCount)--;
            if (imp->refCount <= 0 &&
                imp != q.imp )
                delete imp;

            imp = q.imp;
            (imp->refCount)++;
            return *this;
        }

        ~String()  {
            (imp->refCount)--;
            if (imp->refCount <= 0 ) delete imp;
        }

    private:
        String(char** r) {imp = new StringRep(r);}
        StringRep *imp;
};

```

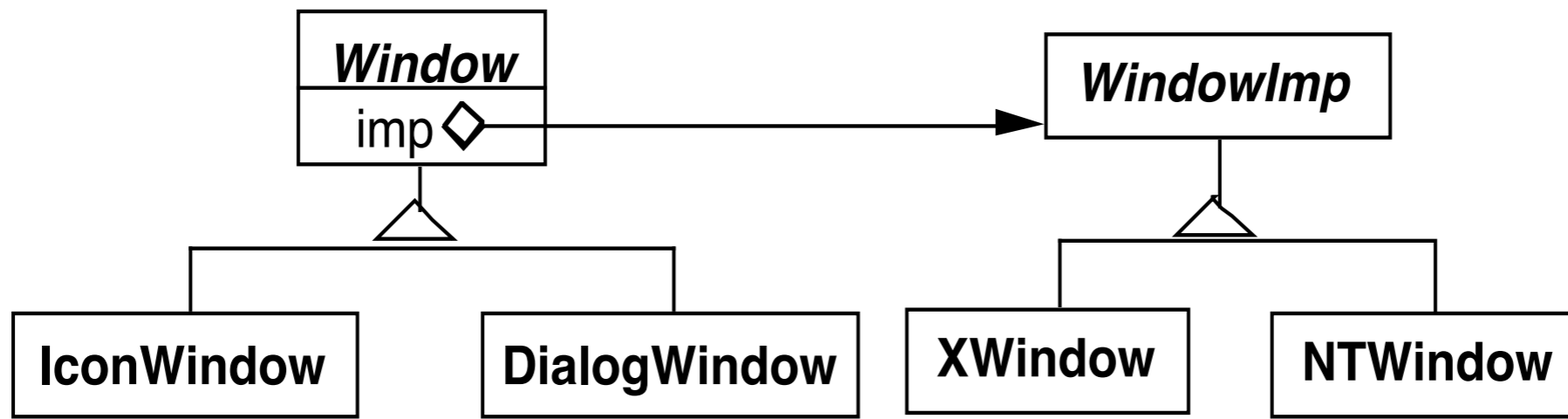
# Why Use Bridge

Implementation selected at run-time

Implementation changed during run-time

# Why Use Bridge

Abstraction & implementations are extensible by subclassing

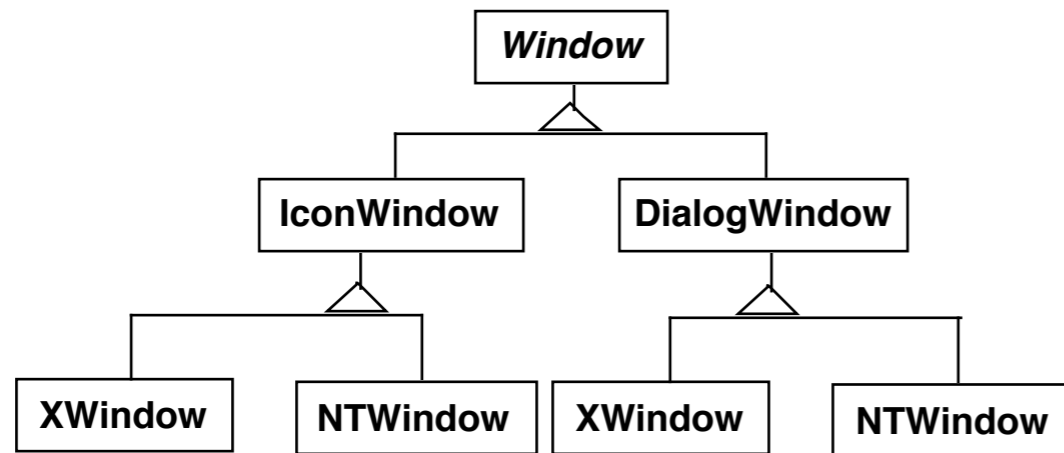


# Why Use Bridge

When changes in the implementation should not require client code to be recompiled

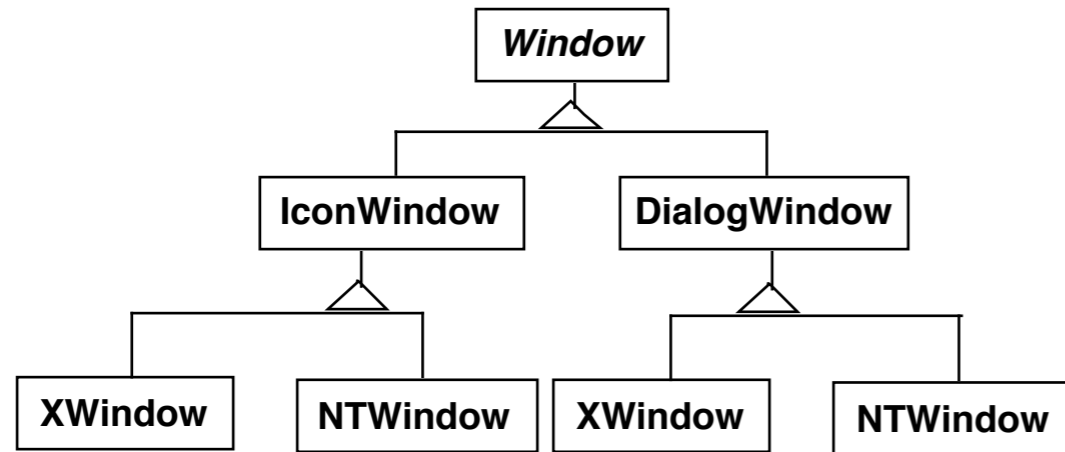
# Why Use Bridge

Proliferation of classes



# Why Use Bridge

Share implementation among multiple objects



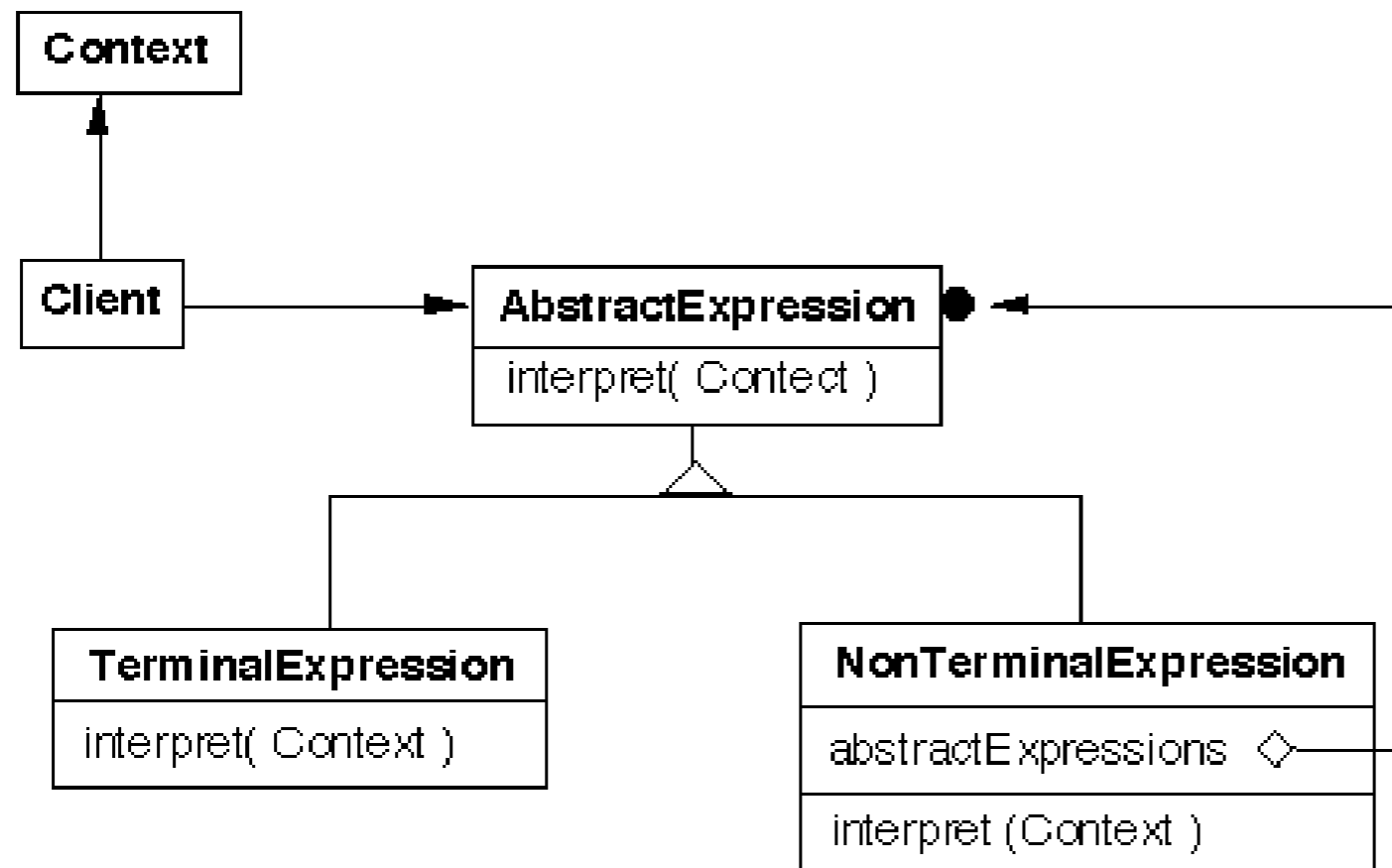
Bridge verses Adapter

Bridge verses Decorator

Bridge & Abstract Factory

# Interpreter

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





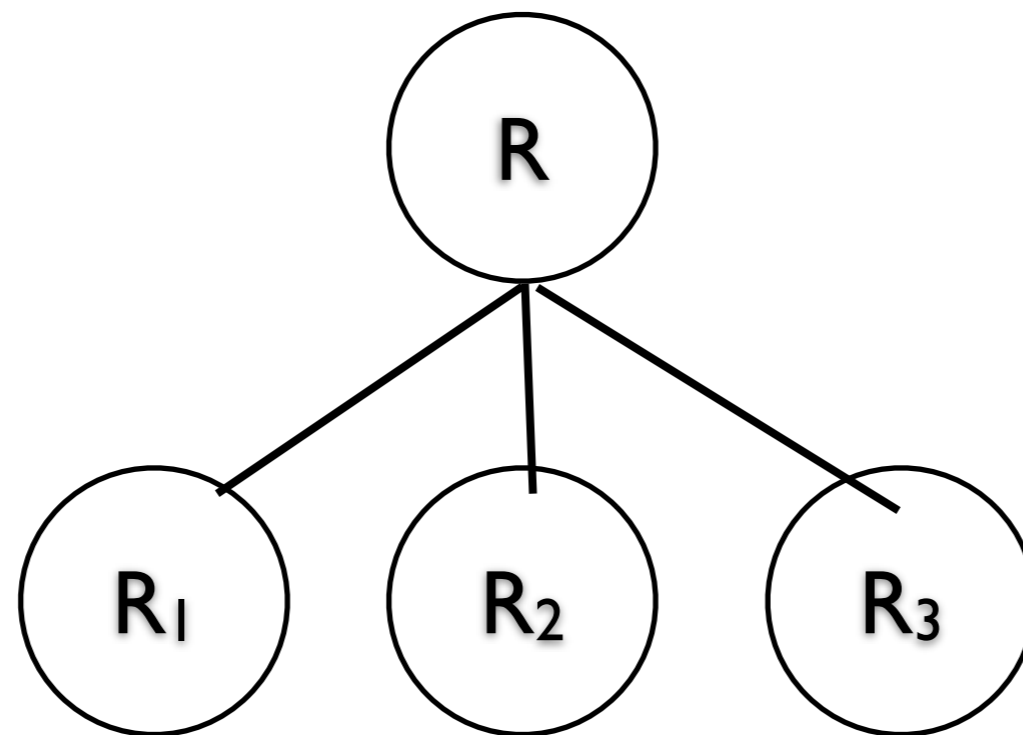
# Grammar & Classes

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 ')'

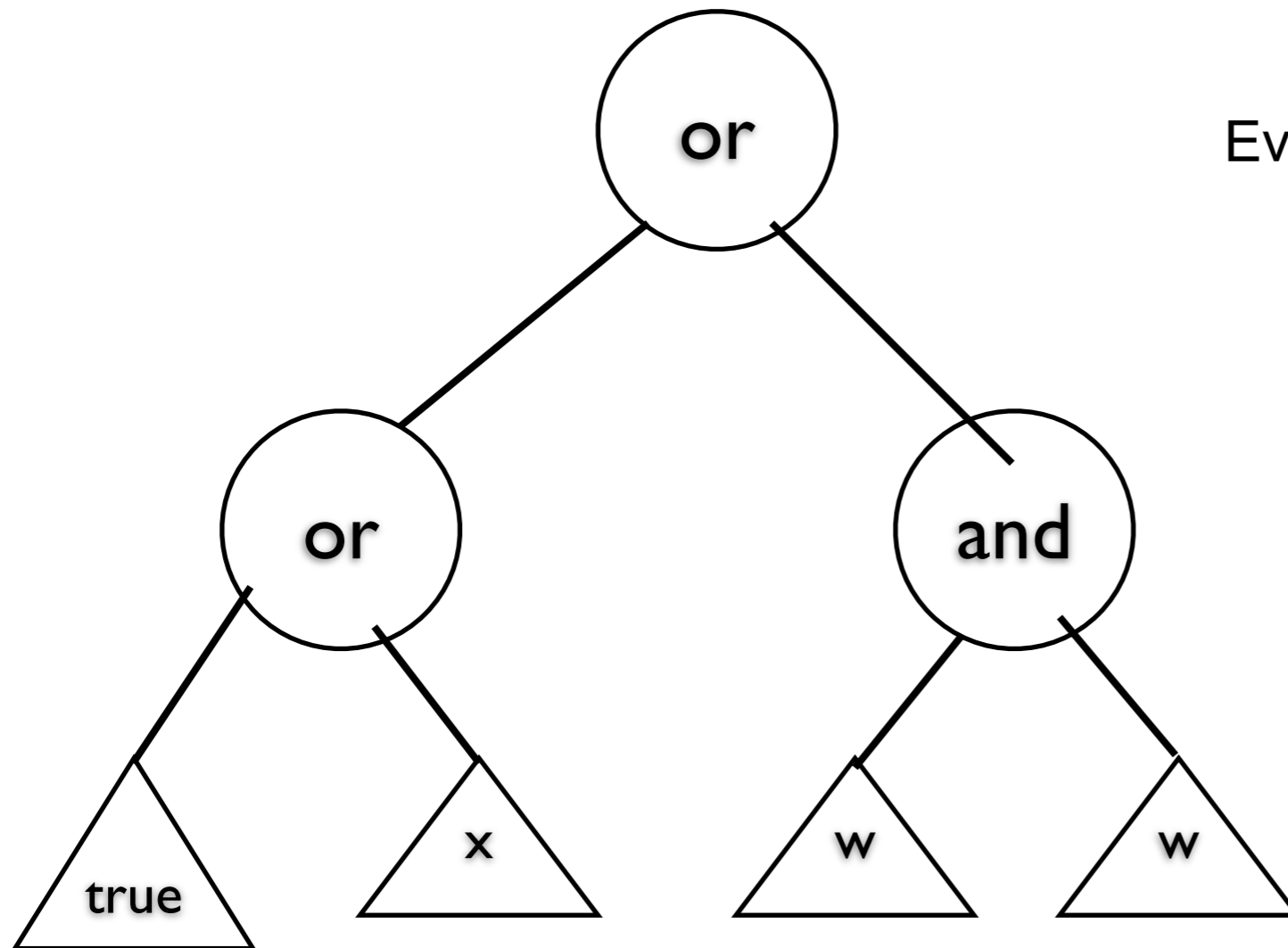
Not ::= 'not' BooleanExpression

Constant ::= 'true' | 'false'

Variable ::= String

# Sample Expression

$((\text{true or } x) \text{ or } (w \text{ and } x))$



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