CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2014
Doc 10 Interpreter, State
Feb 27, 2014

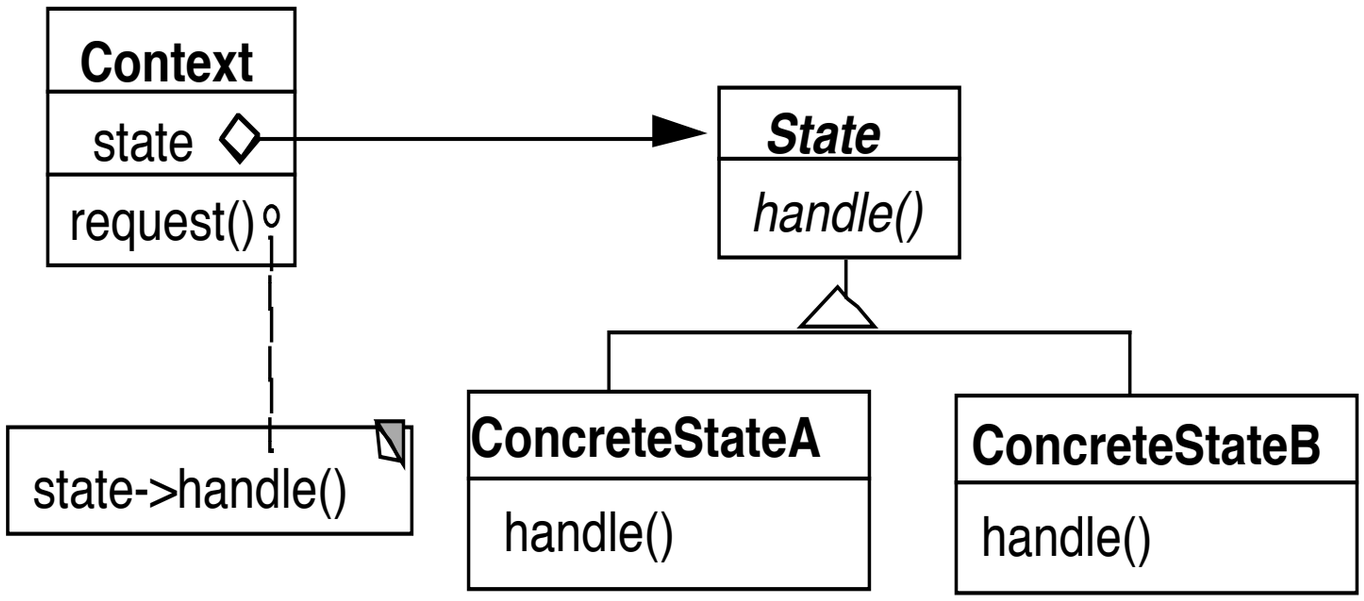
State

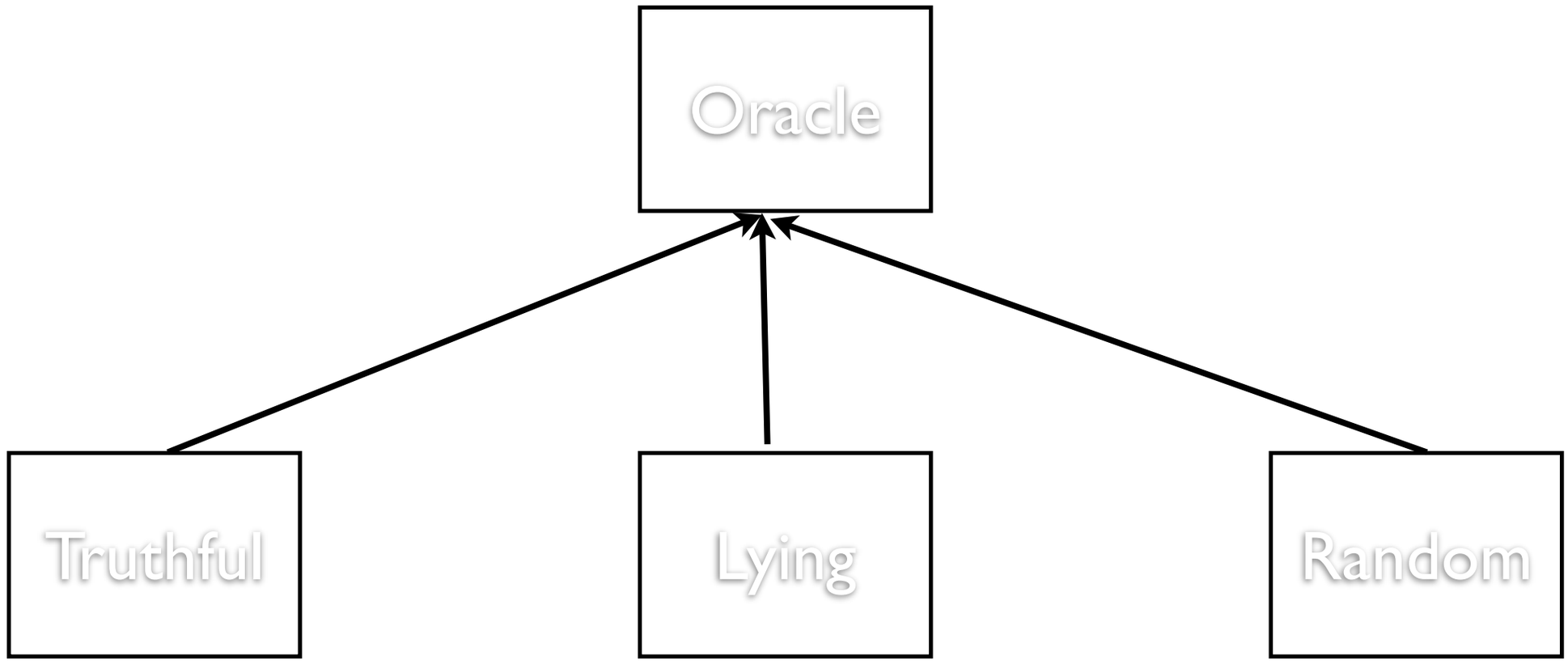
State Pattern

Allow an object to alter its behavior when its internal state changes

The object will appear to change it class

Structure

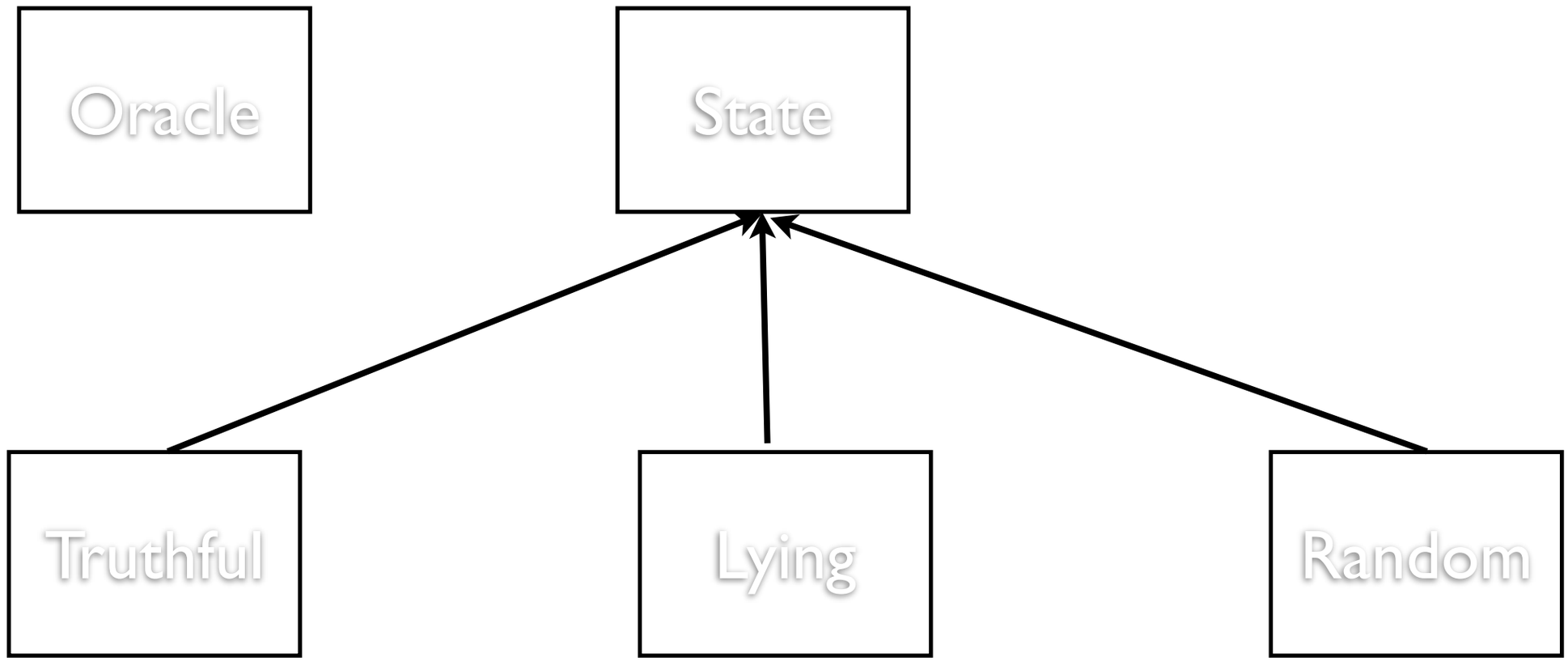

Context
state
request()
State
handle()
state->handle()
ConcreteStateA
handle()
ConcreteStateB
handle()




```
Oracle seer = new Truthful();
seer.willThereBeAFeeIncreaseNextYear();
seer = new Lying();
seer.willThereBeAFeeIncreaseNextYear();
```

```
public class Oracle {
    private final TRUTH = "truth";
    private final LIE = "lie";
    private final RANDOM = "random";

    String state = TRUTH;

    public boolean willThereBeAFeeIncreaseNextYear() {
        if (state == TRUTH)
            blah
        else if (state == LIE)
            more blah
        else if (state == RANDOM)
            random blah
    }
```

Oracle



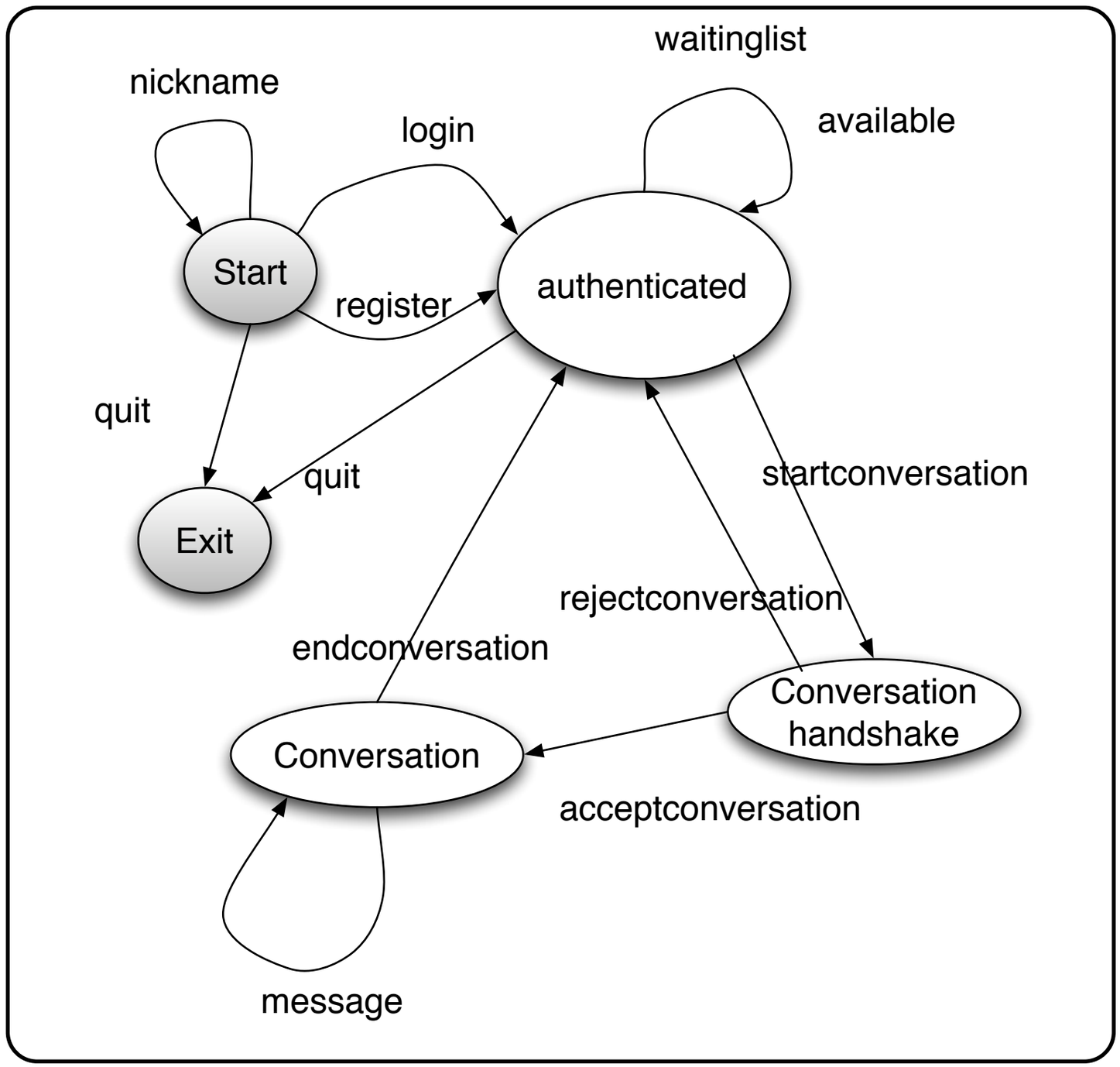

```
class Oracle {
        private State mode = set mode;

        public boolean willThereBeAFeeIncreaseNextYear() {
                return mode. willThereBeAFeeIncreaseNextYear():
        }
}
```

Example: SDChat Server

Commands

"available"
"login"
"register"
"nickname"
"startconversation"
"quit"
"waitinglist"
"acceptconversation"
"message"
"rejectconnection"
"endconversation"

Server States



Without States

```
public class SDChatServer {

    String handleNickname(String data) {
        if (state != START)
            return someErrorMessage();
        handle the main case
    }

    String handleLogin(String data) {
        if (state != START)
            return someErrorMessage();
        handle main case
    }

    String handleWaitinglist(String data) {
        if (state != AUTHENTICATED)
            return someErrorMessage();
        handle main case
    }
```

Who defines state Transitions - Context

```
class Context {
        private AbstractState state = new StartState();

        public Bar foo(int x) {
                int result = state.foo(x);
                if (someConditionHolds() )
                        state = nextState();
        return result;
        }
```

Who defines state Transitions - States

```
class Context {
        private AbstractState state = new StartState();

        public void foo(int x) {
                state = state.foo(x);
                }
```

What if foo returns a value?

Who defines state Transitions - States

```
class Context {
        private AbstractState state = new StartState();

        public int foo(int x) {
                return state.foo(x, this);
                }

        protected void setState(AbstractState newState) {
                state = newState;
        }
```

Sharing State Objects

Stateless state

- State objects without fields
- Can be shared by multiple contexts

Can store date in context and pass as arguments

Large number of state transitions can be expensive

Only create state once & reuse same object

Changing Class - No Need for Context

Language Dependent Feature

Smalltalk & Lisp

```
class Truthful extends Oracle {

    public boolean foo(int x) {
        int result = state.foo(x);
        this.changeClassTo(Random);
        return result;
    }
```

State Verses Strategy

Rate of Change

Strategy

Context usually contains just one strategy object

State

Context often changes state objects

State Verses Strategy

Exposure of Change

Strategy

Strategies all do the same thing

Client do not see change in behavior of Context

State

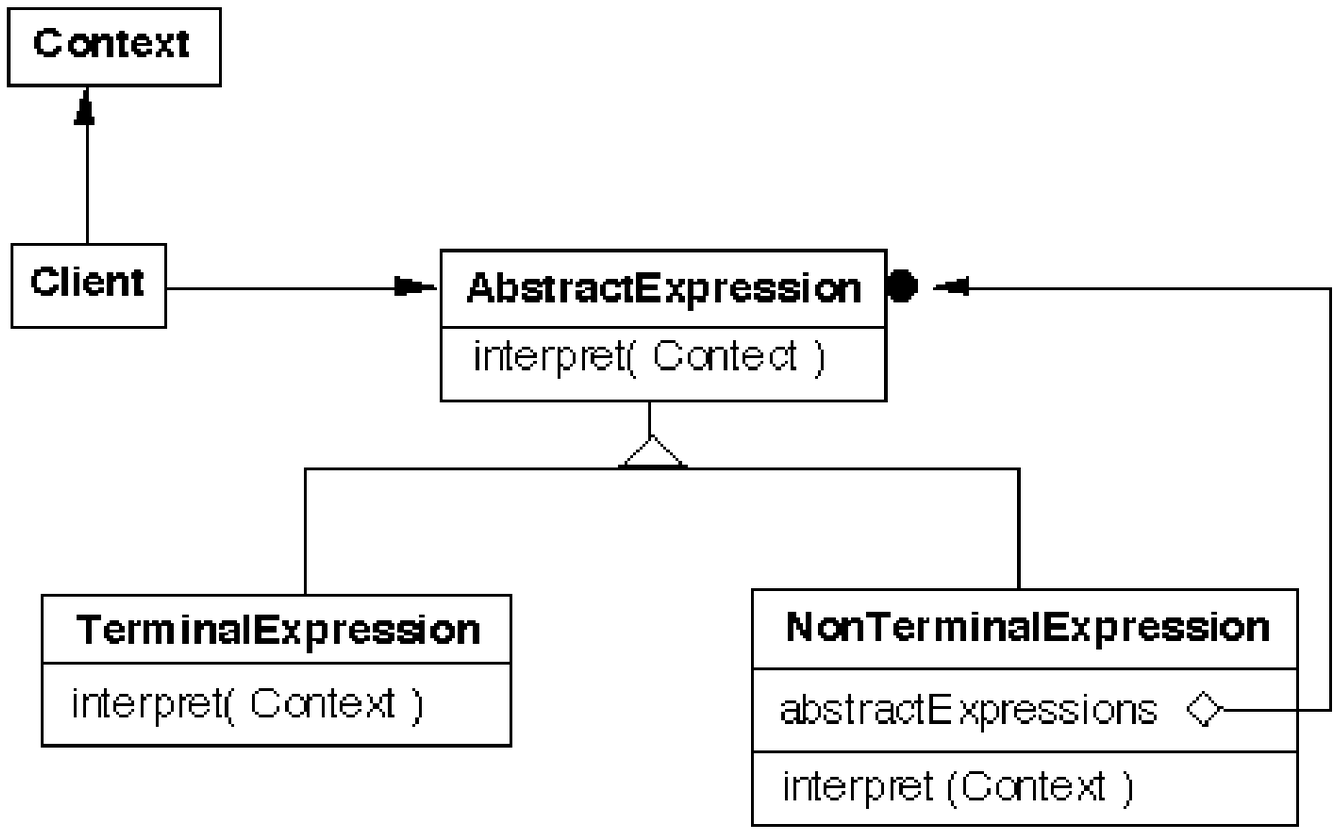
States act differently

Client see the change in behavior

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



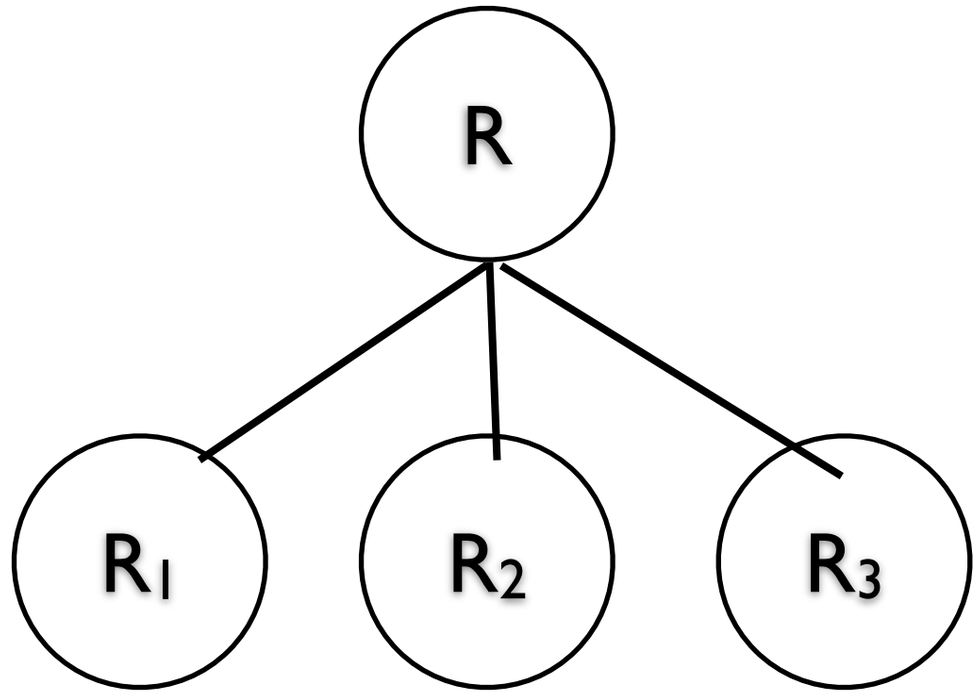
Grammar & Classes

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 ')'

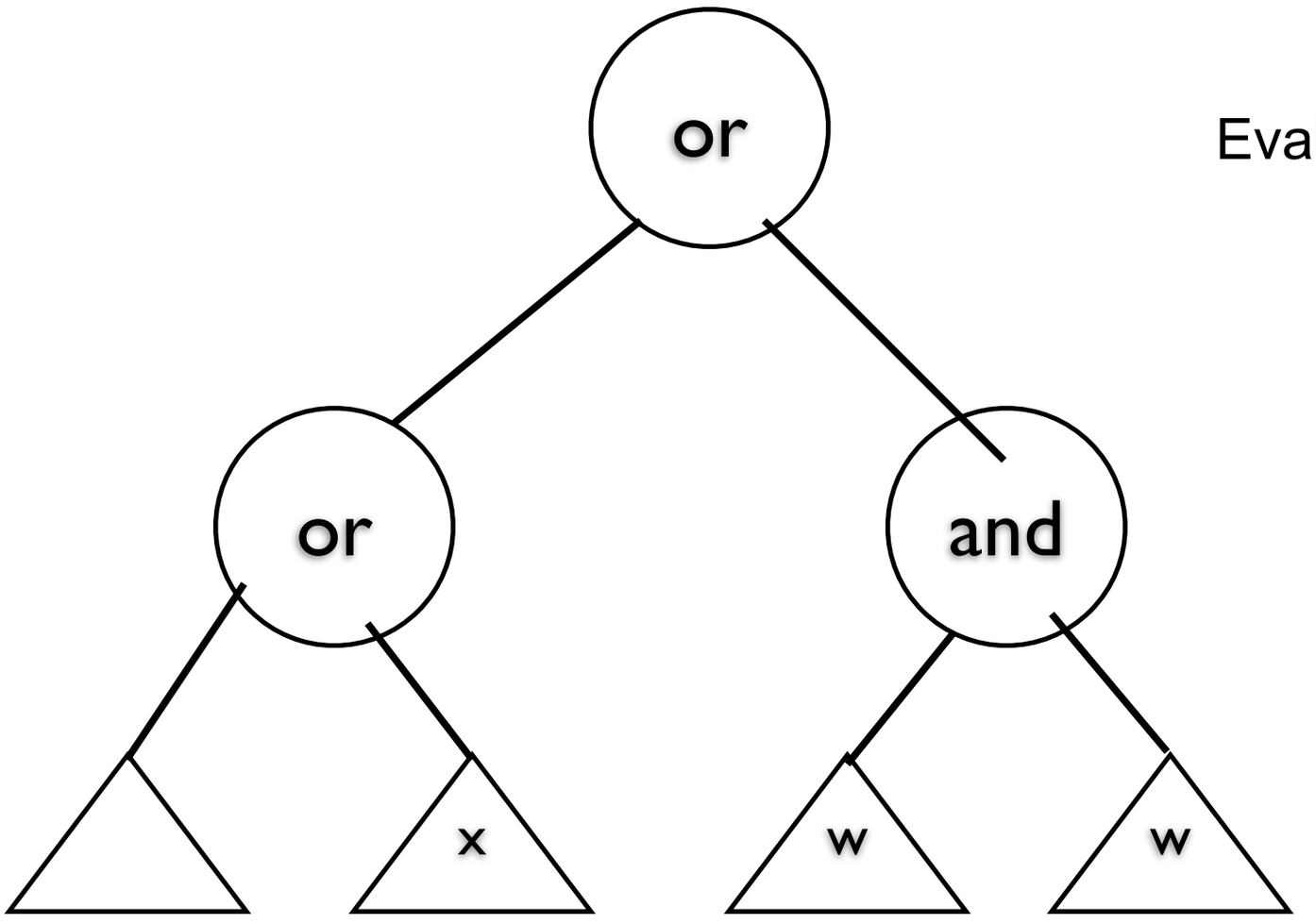
Not ::= ‘not’ BooleanExpression

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