# CS 635 Advanced Object-Oriented Design & Programming
# Spring Semester, 2005
# Doc 13 Interpreter, Strategy & State

**Contents**

## References

Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides, 1995, pp. 243 -256, 315-324, 305-314

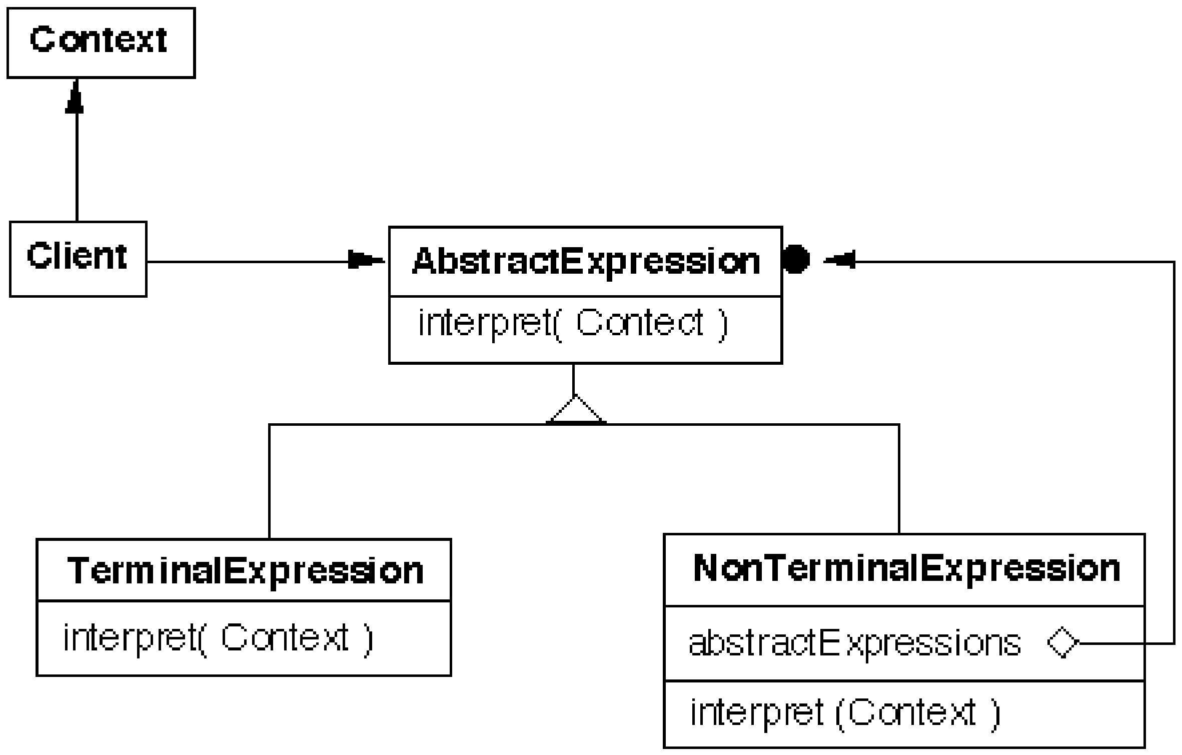
The Design Patterns Smalltalk Companion, Alpert, Brown, Woolf, 1998, pp. 261-272, 339-354, 327-338

Refactoring to Patterns, Kerievsky, 2005, 129-143, 166-177, 269-284

## 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();
}
```

## 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)))
```

## 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 ) );
  }
}
```

## 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

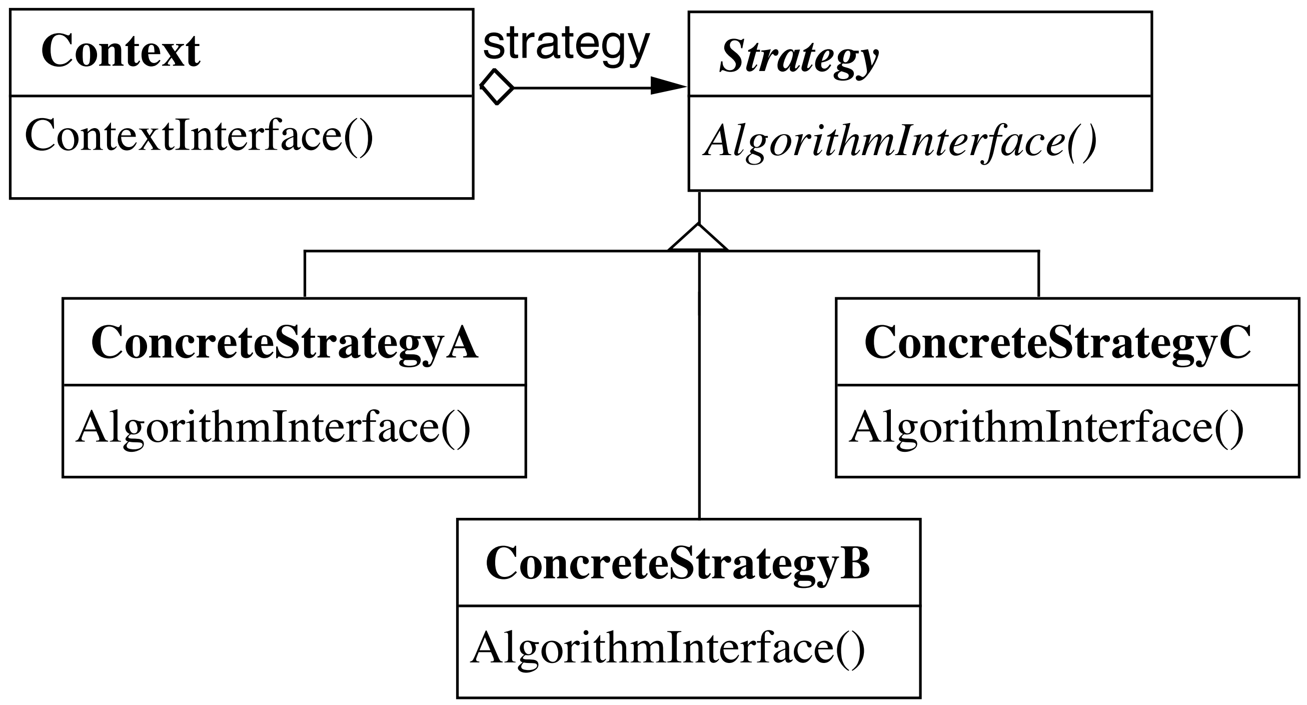
- The above example has each terminal class manage the flyweights for its objects, since Java does limited support for protecting constructors

## Strategy
## Intent

Define a family of algorithms, encapsulate each one, and make them interchangeable

Strategy lets the algorithm vary independently from clients that use it

## Structure



## Examples

Java Layout Managers for Windows

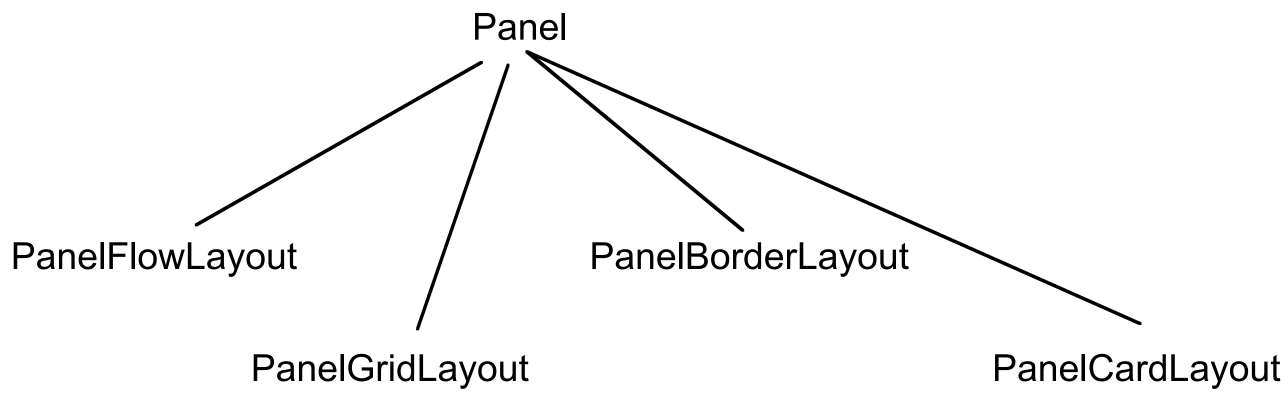
Java Comparators

Smalltalk sort blocks

## Java Layout Managers

```
import java.awt.*;
class  FlowExample  extends Frame  {

  public FlowExample( int  width, int height ) {
     setTitle( "Flow Example" );
     setSize( width, height );
     setLayout( new FlowLayout( FlowLayout.LEFT) );

     for ( int label = 1; label < 10; label++ )
        add( new Button( String.valueOf( label ) ) );
     show();
  }

  public  static  void  main( String  args[] ) {
     new  FlowExample( 175, 100 );
     new  FlowExample( 175, 100 );
  }
}
```

## Why Not use Inheritance?



But there are:

- 20 different Layout classes
- At least 39 subclasses of Component using layouts

So using inheritance would require 780 classes!

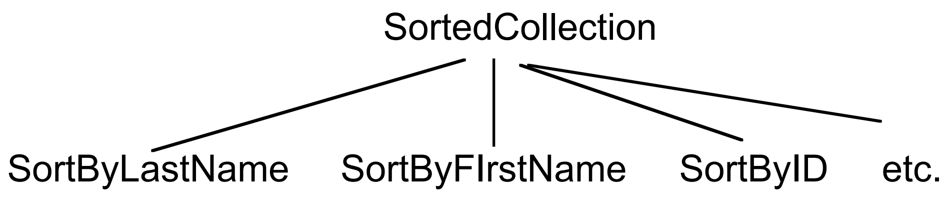
## Java Comparators

```
import java.util. Comparator;
import java.util.*;

class Student  {
   String name;

   public Student( String newName) { name = newName;}

   public String toString() { return name; }
   }

final class StudentNameComparator implements Comparator {

   public int compare( Object leftOp, Object rightOp ) {
      String leftName = ((Student) leftOp).name;
      String rightName = ((Student) rightOp).name;
      return leftName.compareTo( rightName );
   }

   public boolean equals( Object comparator ) {
      return comparator instanceof StudentNameComparator;
   }
}

public class Test  {
   public static void main(String args[])  {
      Student[] cs596 = { new Student( "Li" ), new Student( "Swen" ),
           new Student( "Chan" ) };
      //Sort the array
      Arrays.sort( cs596, new StudentNameComparator() );
   }
}
```

## Smalltalk SortBlocks

```
| list |
list := #( 1 6 2 3 9 5 ) asSortedCollection.
Transcript
  print: list;
  cr.
list sortBlock: [:x :y | x > y].
Transcript
  print: list;
  cr;
  flush.
```

## Why Not use Inheritance



There are arbitrarily many ways to sort

So get arbitrarily many

- Subclasses of SortedCollection or
- Comparator classes (blocks)

But with comparators (blocks) one can:

- Combine different comparators
- Sort the same list with different comparators

## Applicability

Use the Strategy pattern when

- You need different variants of an algorithm
- An algorithm uses data that clients shouldn't know about
- A class defines many behaviors, and these appear as multiple switch statement in the classes operations
- Many related classes differ only in their behavior

## Consequences

- Families of related algorithms
- Alternative to subclassing of Context

  What is the big deal? You still subclass Strategy!

- Eliminates conditional statements

  Replace in Context code like:

```
switch  ( flag ) {
   case A: doA(); break;
   case B: doB(); break;
   case C: doC(); break;
}
```

  With code like:

```
strategy.do();
```

- Gives a choice of implementations
- Clients must be aware of different Strategies

```
SortedList studentRecords = new SortedList(new ShellSort());
```

- Communication overhead between Strategy and Context
- Increase number of objects

## Implementation

- Defining the Strategy and Context interfaces

  How does data flow between them

  Context pass data to Strategy

  Strategy has point to Context, gets data from Context

  In Java use inner classes

- Strategies as template parameters

  Can be used if Strategy can be selected at compile-time and does not change at runtime

```
        SortedList<ShellSort> studentRecords;
```

- Making Strategy objects optional

  Give Context default behavior

  If default used no need to create Strategy object

## State
## Example - SPOP
## Simple Post Office Protocol

SPOP is used to download e-mail from a server

SPOP supports the following command:

- USER <username>
- PASS <password>
- LIST
- RETR <message number>
- QUIT

## USER & PASS Commands

USER with a username must come first
PASS with a password or QUIT must come after USER

If the username and password are valid, then the user can use other commands

## LIST Command

Arguments: a message-number (optional)

If it contains an optional message number then returns the size of that message

Otherwise return size of all mail messages in the mailbox

## RETR Command

Arguments: a message-number

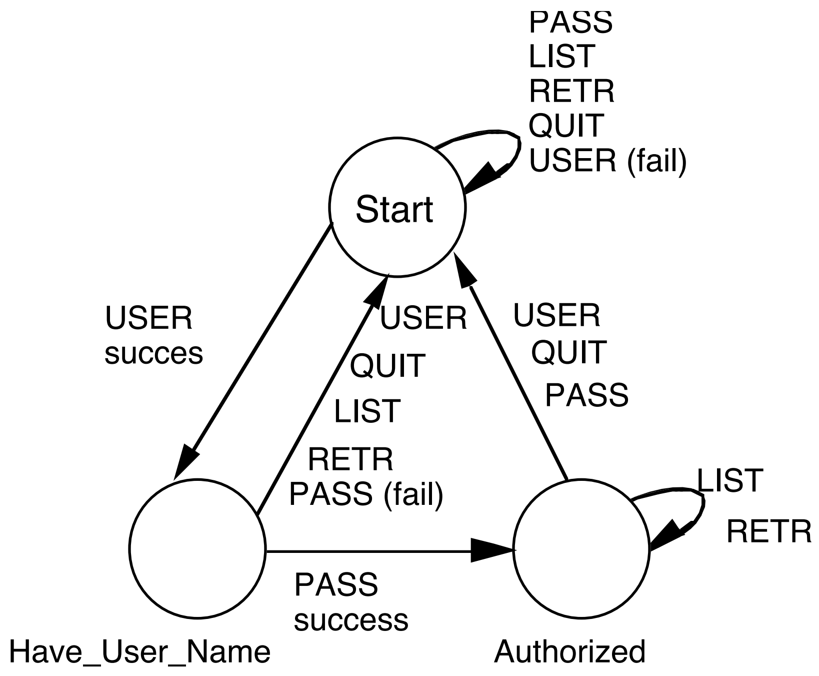
Returns: the mail message indicated by the number

## QUIT Command

Arguments: none

Updates mail box to reflect transactions taken during the transaction state, then logs user out

If session ends by any method except the QUIT command, the updates are not done



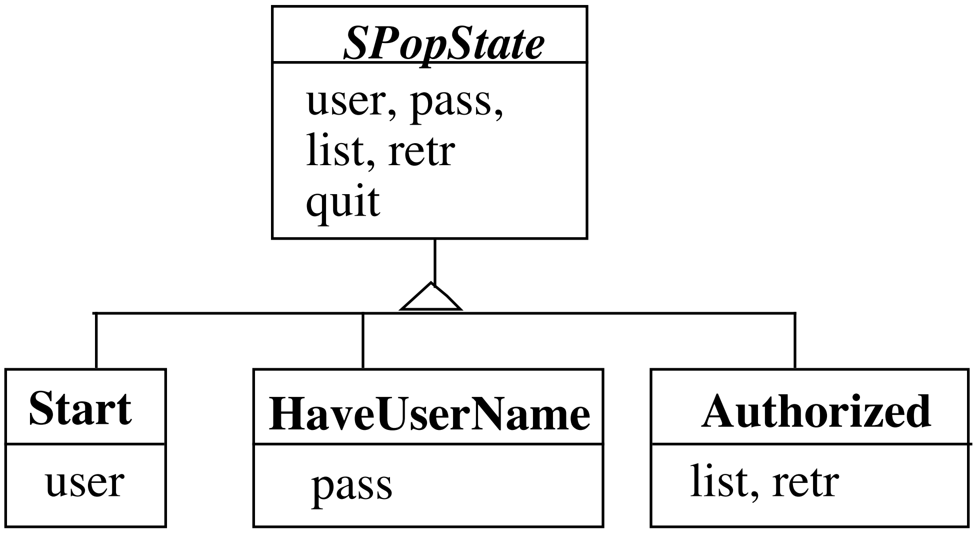
## The Switch Statement

```
class SPop
  {
  static final int HAVE_USER_NAME = 2;
  static final int START = 3;
  static final int AUTHORIZED = 4;

  private int state = START;

  String userName;
  String password;

  public void user( String userName ) {
     switch (state) {
        case START:  {
           this.userName = userName;
           state = HAVE_USER_NAME;
           break;
        }
        case HAVE_USER_NAME:
        case AUTHORIZED:{
           endLastSessionWithoutUpdate();
           goToStartState()
           }
        }
     }
```

## Implementation with Switch Statement Cont.

```
public void pass( String password )
   {
   switch (state)
      {
      case START:  {
         giveWarningOfIllegalCommand();
         }
      case HAVE_USER_NAME: {
         this.password = password;
         if ( validateUser() )
            state = AUTHORIZED;
         else {
            sendErrorMessageOrWhatEver();
            userName = null;
            password = null;
            state = START;
         }
      case AUTHORIZED: {
         endLastSessionWithoutUpdate();
         goToStartState()
      }
      }
   }
etc.
}
```

## Using Polymorphism Implementation



```
class SPop {
  private SPopState state = new Start();

  public void user( String userName ) {
     state = state.user( userName );
  }

  public void pass( String password ) {
     state = state.pass( password );
  }

  public void list( int messageNumber ) {
     state = state.list( massageNumber );
  }

  etc.
```

## SPopStates

Defines default behavior

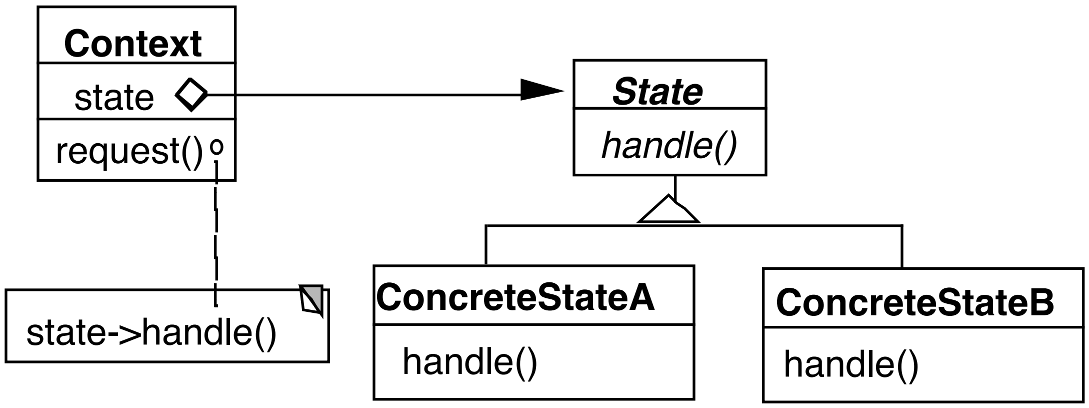
```
abstract class SPopState {
  public SPopState user( String userName ) {
    return goToStartState();
  }

  public SPopState pass( String password ) {
    return goToStartState();
  }

  public SPopState list( int massageNumber ) {
    return goToStartState();
  }

  public SPopState retr( int massageNumber ) {
    return goToStartState();
  }

  public SPopState quit(  ) {
    return goToStartState();
  }

  protected SPopState goToStartState()  {
    endLastSessionWithoutUpdate();
    return new StartState();
  }
}
```

## SpopStates - Continued

```
class Start extends SPopState {
  public SPopState user( String userName ) {
    return new HaveUserName( userName );
  }
}

class HaveUserName extends SPopState {
  String userName;

  public HaveUserName( String userName ) {
    this.userName = userName;
  }

  public SPopState pass( String password ) {
    if ( validateUser( userName, password )
      return new Authorized( userName );
    else
      return goToStartState();
  }
}
```

## State
## Intent

Allow an object to alter its behavior when its internal state changes. The object will appear to change it class.



## Applicability

Use the State pattern in either of the following cases:

- An object's behavior depends on its state, and it must change its behavior at run-time depending on that state.
- Operations have large, multipart conditional statements that depend on the object's state. Often, several operations will contain this same conditional structure.

## Issues
## How much State in the State

In Example:

- SPop is the Context
- SPopState is the abstract State
- Start, HaveUserName are ConcreteStates

All the state & all real behavior is in SPopState & subclasses

This is an extreme example

In general the Context will have data & methods
- Besides State & State methods
- This data will not change states

That is only some aspects of the Context will alter its behavior

## Issue
## Who defines the state transitions?
## The Context

- If the states will be used in different state machines with different transitions
- If the criteria changing states is fixed

```
class SPop
  {
  private SPopState state = new Start();

  public void user( String userName )
    {
    state.user( userName );
    state = new HaveUserName( userName );
    }

  public void pass( String password )
    {
    if ( state.pass( password ) )
      state = new Authorized( );
    else
      state = new Start();
    }
```

## Who defines the state transitions?
## The State

- More flexible to let State subclasses specify the next state

```
class SPop
  {
  private SPopState state = new Start();

  public void user( String userName )
    {
    state = state.user( userName );
    }

  public void pass( String password )
    {
    state = state.pass( password );
    }

  public void list( int messageNumber )
    {
    state = state.list( massageNumber );
    }
```

## Issue

## Sharing State Objects

Multiple contexts (SPops) can use the same state object if the state object has no instance variables

A state object can have no instance variables if:

- The object has no need for instance variables or
- The object stores its instance variables elsewhere

## Storing Instance Variables Elsewhere
## Variant 1

SPop stores them and passes them to states

```
class SPop
  {
  private SPopState state = new Start();

  String userName;
  String password;

  public void user( String newName )
     {
     this.userName = newName;
     state.user( newName );
     }

  public void pass( String password )
     {
     state.pass( userName , password );
     }
```

## Storing Instance Variables Elsewhere
## Variant 2

SPop stores them and states get data from SPop

```
class SPop {
  private SPopState state = new Start();

  String userName;
  String password;

  public String userName() { return userName; }

  public String password() { return password; }

  public void user( String newName ) {
     this.userName = newName ;
     state.user( this );
  }

  etc.
```

```
class HaveUserName extends SPopState {
  public SPopState pass( SPop mailServer ) {
     String useName = mailServer.userName();
     etc.
  }
}
```

## Issue
## Creating and Destroying State Objects

Options:

- Create state object when needed, destroy it when it is no longer needed
- Create states once, never destroy them (singleton)

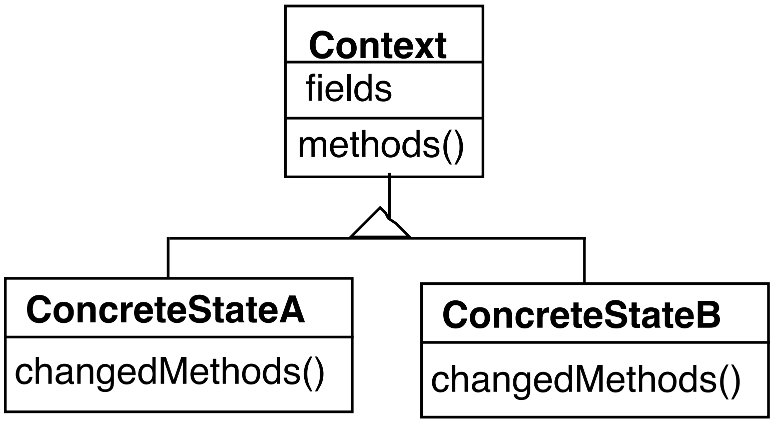
## Issue
## Changing the Context Class for Real

Some languages allow an object to change its class

- CLOS (Common Lisp Object System)
- Cincom's VisualWorks Smalltalk

```
| context |
context := Start new.
context changeClassTo: HaveUserName.
context changeClassTo: Authorized.
```

So why not forget State pattern and use:



In VisualWorks Smalltalk
- Problems arise if ConcreteStates have fields

In CLOS the State pattern may not be needed

## Consequences

- It localize state-specific behavior and partitions for different states
- It makes state transitions explicit
- State objects can be shared
- Complicates a design when state-changing logic is already easy to follow

## State Verses Strategy

How to tell the difference

### Rate of Change

Strategy

- Context object usually contains one of several possible ConcreteStrategy objects

State

- Context object often changes its ConcreteState object over its lifetime

### Exposure of Change

Strategy

- All ConcreteStrategies do the same thing, but differently

- Clients do not see any difference in behavior in the Context

State

- ConcreteState act differently

- Clients see different behavior in the Context