CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2004
Doc 10 State & Chain of Responsibility

Contents

References

Design Patterns: Elements of Resuable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides, Addison Wesley, 1995, pp. 305-314, 223-232

The Design Patterns Smalltalk Companion, Alpert, Brown, Woolf, Addision-Wesley, 1998, pp. 327-338, 225-245

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

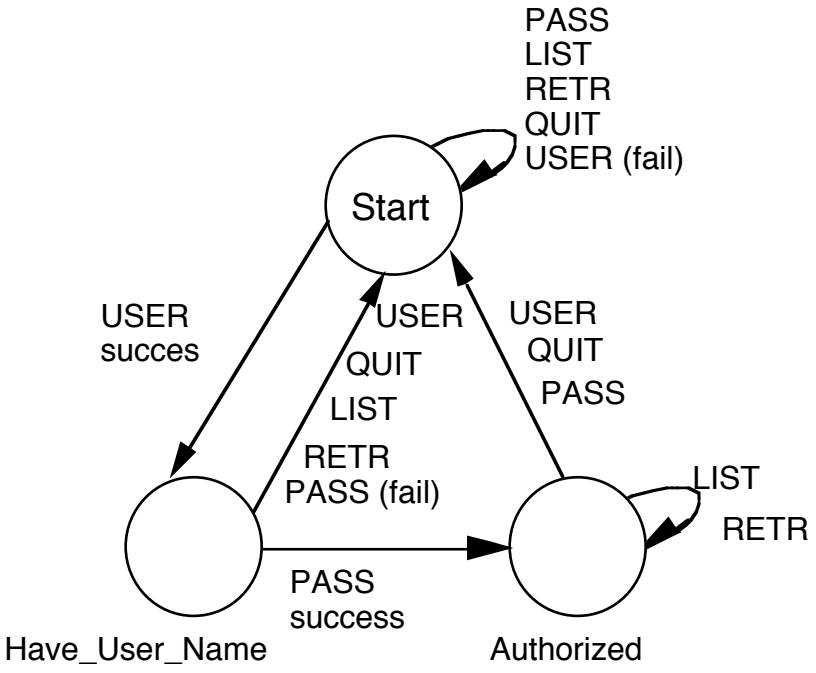
Returns: the mail message indicated by the number

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



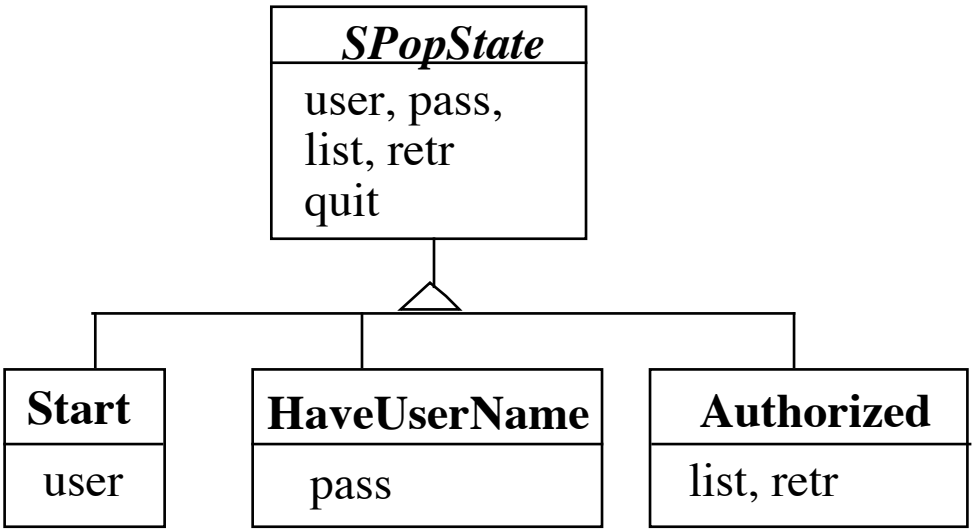
The Switch Statement

```
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();
   }
}
```

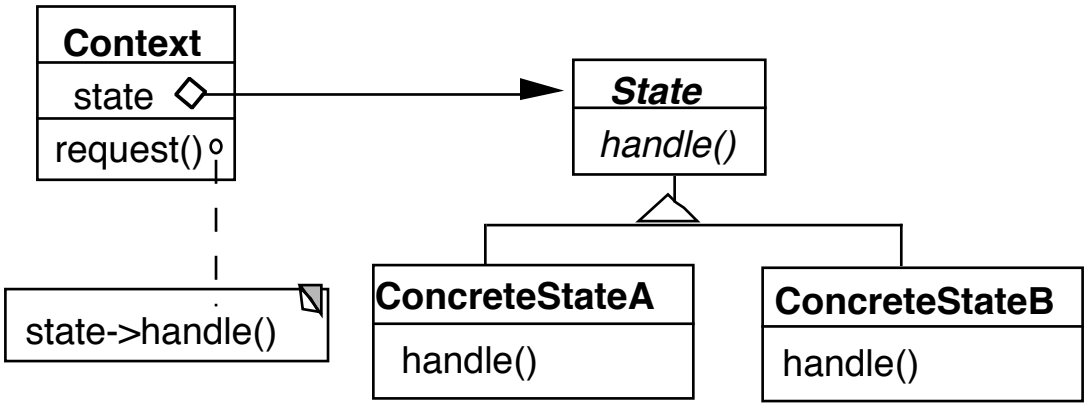
SpopStates - Continued

```
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)

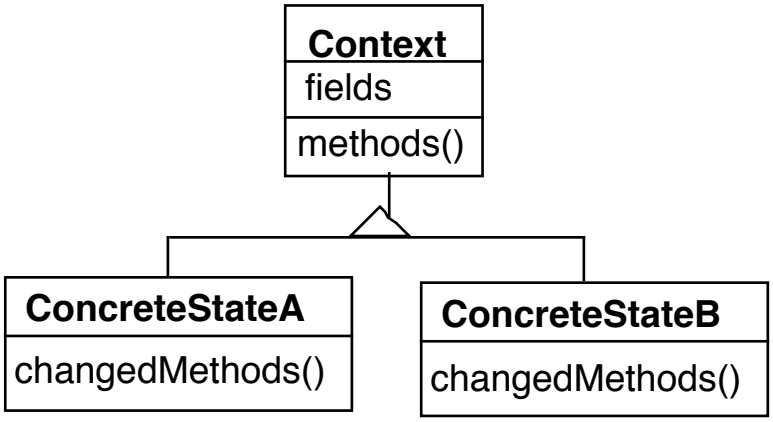
Issue
Changing the Context Class for Real

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

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

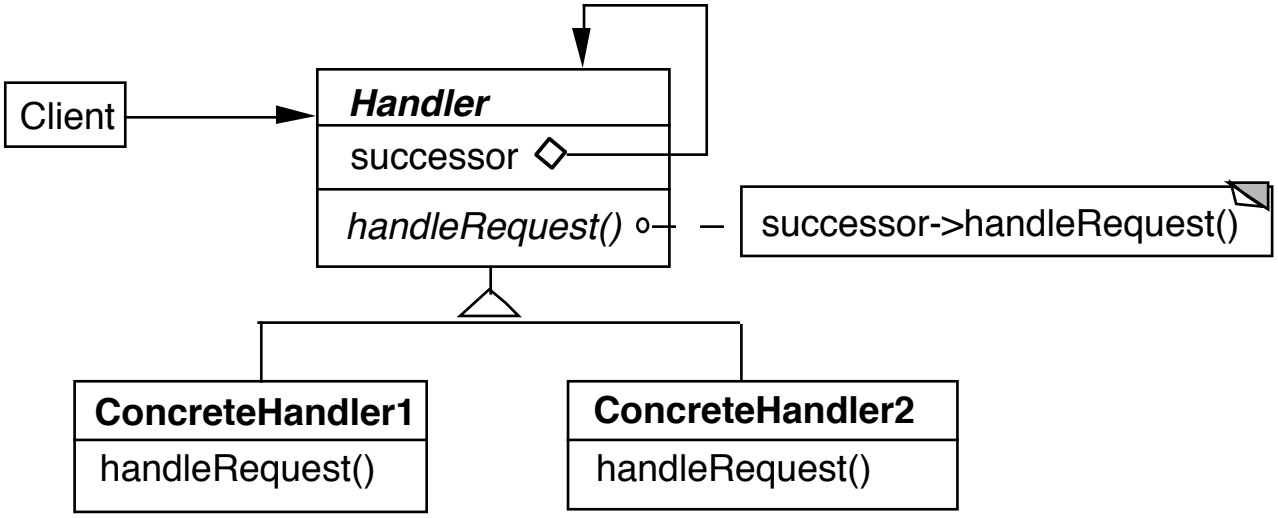
Strategy
- All ConcreteStrategies do the same thing, but differently
- Clients do not see any difference in behavior in the Context

State
- ConcreteState act differently
- Clients see different behavior in the Context

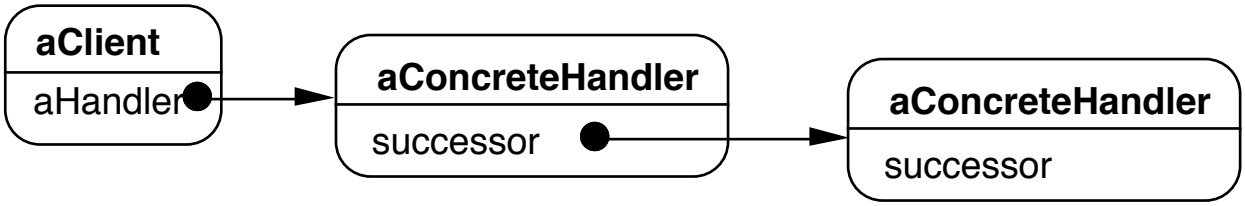
Chain of Responsibility
Intent

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Class Structure



Sample Object Structure



Participants

Handler

- Defines the interface for handling the requests
- May implement the successor link

ConcreteHandler

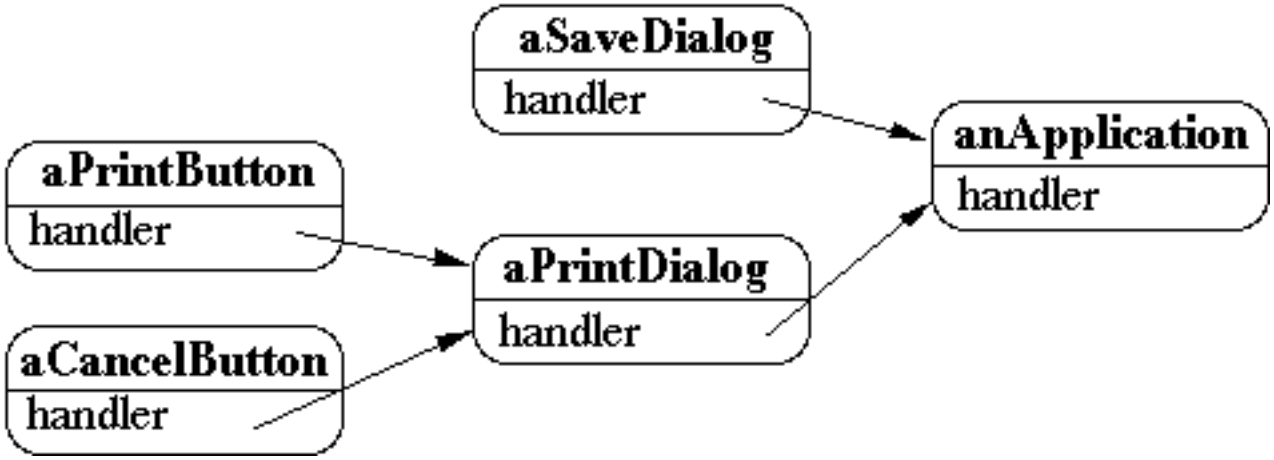
- Handles requests it is responsible for
- Can access its successor
- Handles the request if it can do so, otherwise it forwards the request to its successor

Consequences

- Reduced coupling
- Added flexibility in assigning responsibilities to objects
- Not guaranteed that request will be handled

Motivation

Context Help System



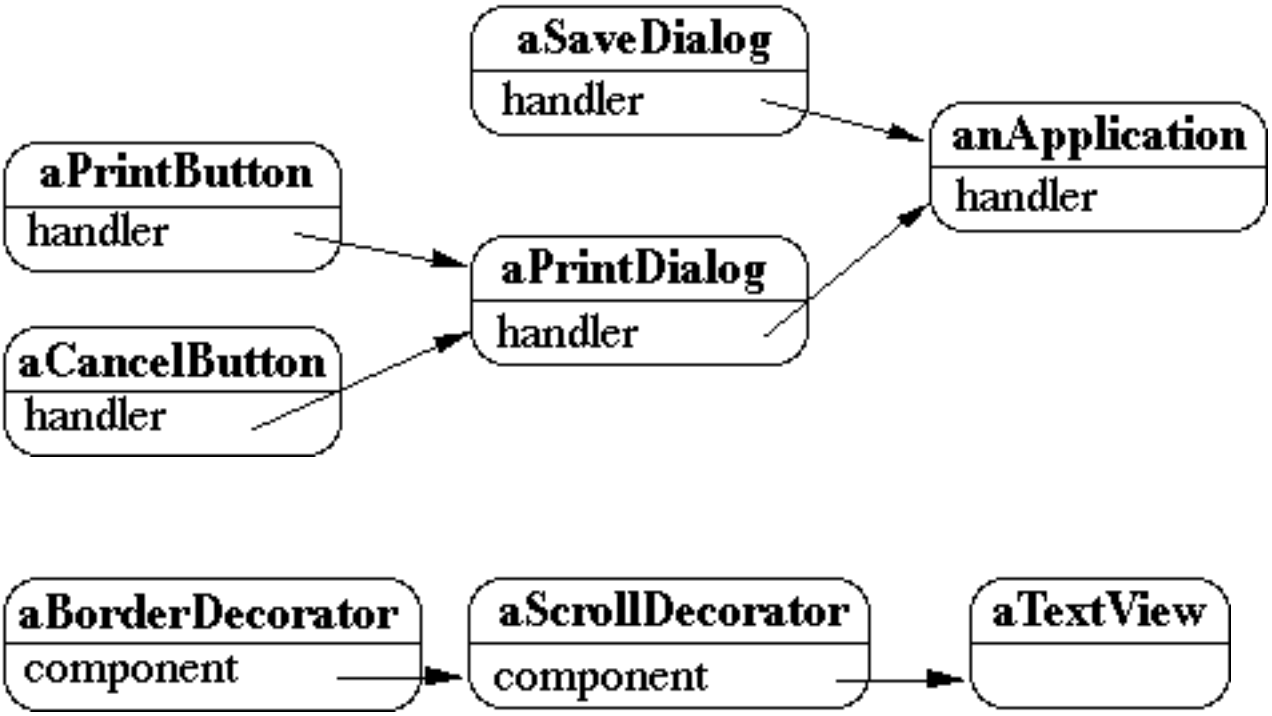
When to Use

When more than on object may handle a request, and the handler isn't known a priori

When you want to issue a request to one of several objects without specifying the receiver explicitly

When the set of objects that can handle a request should be specified dynamically

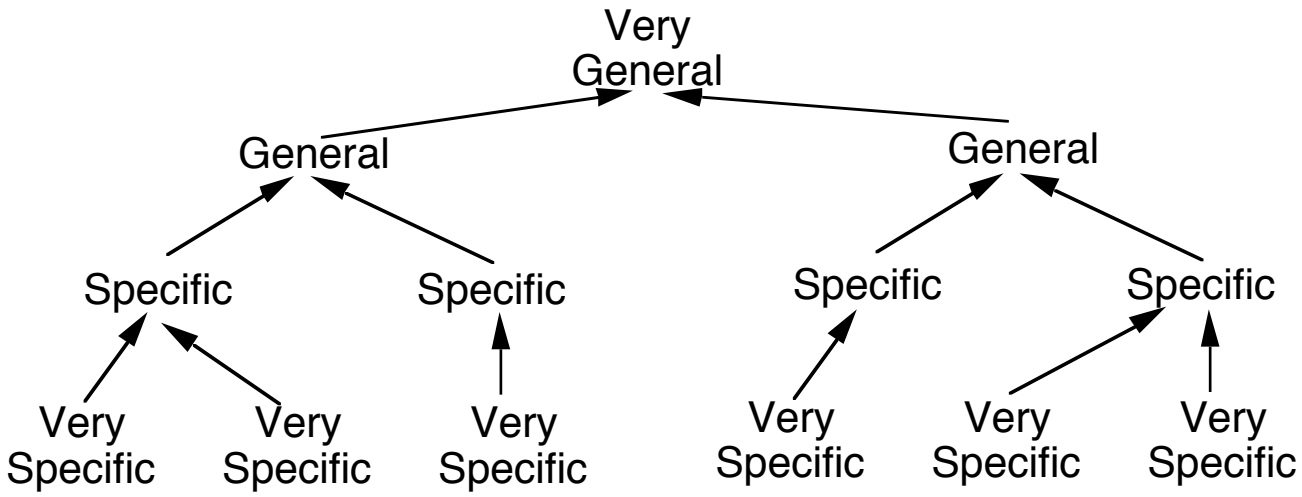
How does this differ from Decorator?



Chain of Command

Like the military

- A request is made
- It goes up the chain of command until someone has the authority to answer the request



Implementation Issues
The successor chain

- Use existing links in the program

  The concrete handlers may already have pointers to their successors, so just use them

- Define new links

  Give each handler a link to its successor

Representing Requests

- Each request can be a hard-coded

```
abstract class HardCodedHandler
  {
  private HardCodedHandler successor;

  public HardCodedHandler( HardCodedHandler aSuccessor)
    { successor = aSuccessor; }

  public void handleOpen()
    { successor.handleOpen(); }

  public void handleClose()
    { successor.handleClose(); }

  public void handleNew( String fileName)
    { successor.handleClose( fileName ); }
  }
```

Representing Requests

- A single method implements all requests

```
abstract class SingleHandler {
   private SingleHandler successor;

   public SingleHandler( SingleHandler aSuccessor) {
      successor = aSuccessor;
   }

   public void handle( String request) {
      successor.handle( request );
   }
}

class ConcreteOpenHandler extends SingleHandler {
   public void handle( String request) {
      switch ( request ) {
         case "Open" : do the right thing;
         case "Close" : more right things;
         case "New" : even more right things;
         default: successor.handle( request );
      }
   }
}
```

Representing Requests

- Single handle method with Request Object for parameters

```
abstract class SingleHandler {
   private SingleHandler successor;

   public SingleHandler( SingleHandler aSuccessor)
      {successor = aSuccessor; }

   public void handle( Request data)
      { successor.handle( data ); }
}

class ConcreteOpenHandler extends SingleHandler {
   public void handle( Open data)
      { // handle the open here }
}

class Request {
   private int size;
   private String name;
   public Request( int mySize, String myName)
      { size = mySize;  name = myName; }

   public int size() { return size; }
   public String name() { return name;}
}

class Open extends Request
   {// add Open specific stuff here}

class Close extends Request
   { // add Close specific stuff here}
```

Object-Oriented Recursion
Recursive Delegation

A method polymorphically sends its message to a different receiver

Eventually a method is called that performs the task

The recursion then unwinds back to the original message send

Example

```
class BinarySearchTree {
   Node root

   boolean containsKey( Object key ) {
      return root.containsKey(key);
   }

   String toString() {
      return "Tree( " + root.toString() + ")";
   }
   blah
}
```

Example Continued

```
class BinaryNode implements Node {
   Node left;
   Node right;
   Object key;
   Object value;

   boolean containsKey( Object key ) {
      if this.key == key
         return true;
      if this.key < key
         return right.containsKey(key);
      if this.key > key
         return left.containsKey( key);
   }

   String toString() {
      return "( " + left.toString() + key + right.toString() + ")";
   }
   blah
}

class NullNode implements Node {

   boolean containsKey( Object key ) {
      return false;
   }

   String toString() {
      return " ";
   }
   blah
}
```