CS 580 Client-Server Programming
Fall Semester, 2002
Doc 20 Some Parsing & States
Contents

References

Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides, Addison-Wesley, 1995

Past CS580 lecture notes

Server Structure

Servers perform a number of standard operations

- Handling sockets
- Reading/Writing to sockets
- Handling threads
- Parsing messages
- Logging
- Setting parameters

How some operations are done are

- The same on different servers

- Differ between servers

Isolate these operations to make it easier to

- Implement the operation
- Test the operations
- Build a server
- Reuse parts in other servers
- Reuse structure

Parsing & Messages

A client sends a request to a server

The server

- Performs some action
- Sends the client a response

Yet we have:

```
InputStream in = aSocket.getInputStream();
Stringbuffer message = new StringBuffer();
while (some end condition)
   {
   int c = in.read();
   if (c != -1)
      message.append( (char) c );
   }
```

Why not

```
PopReadStream in = new PopStream( aSocket);
PopRequest clientRequest = in.read();
```

Request & Response Classes

Request & Response classes for your protocol:

Isolate the syntactic details of the protocol

Allow client & server to deal with higher-level structures

aResponse.toString()
aResponse printString
Format the response according to the protocol

Given a message string in the protocol create correct object

```
new PopResponse( aPopMessageString );
PopResponse fromString: aPopMessageString)
```

What other operations should they have?

High-level Streams

Create Stream classes that read/write messages not chars

So client does:

```
PopWriteStream out = new PopWriteStream( aSocket);
PopReadStream in = new PopReadStream( aSocket);

PopRequest user = PopRequest.user( “whitney”);
out.write( user);
PopResponse result = in.read();
If (result.isSuccess() )
   {
   out.write( PopRequest.pass( “I forget”));
   }
```

Server does

```
clientIO := PopReadWriteStream on: aSocket
request := clientIO next.
request isPassword
   ifFalse: [clientIO nextPut: (PopResponse error)]
```

POP & Parsing
POP Requests

Very regular & easy to parse

keyword blank argument$_1$ [blank argument$_k$] CRLF

```
crlf := String with: Character cr with: Character lf.
mesageString := inputStream upToAndSkipThroughAll: crlf.
messageParts := messageString tokensBasedOn: Character cr.
```

```
messageString = aBufferedReader.readline();
messageParts = messageString.split( " ");
```

POP Responses

Two different formats

Status keyword additionalInfo CRLF

Status keyword additionalInfo CRLF
Line1 CRLF
Line2 CRLF
etc
LineN CRLF
.CRLF

How do you know which format you are reading?

States

Some Servers are stateful

Each connection has different states

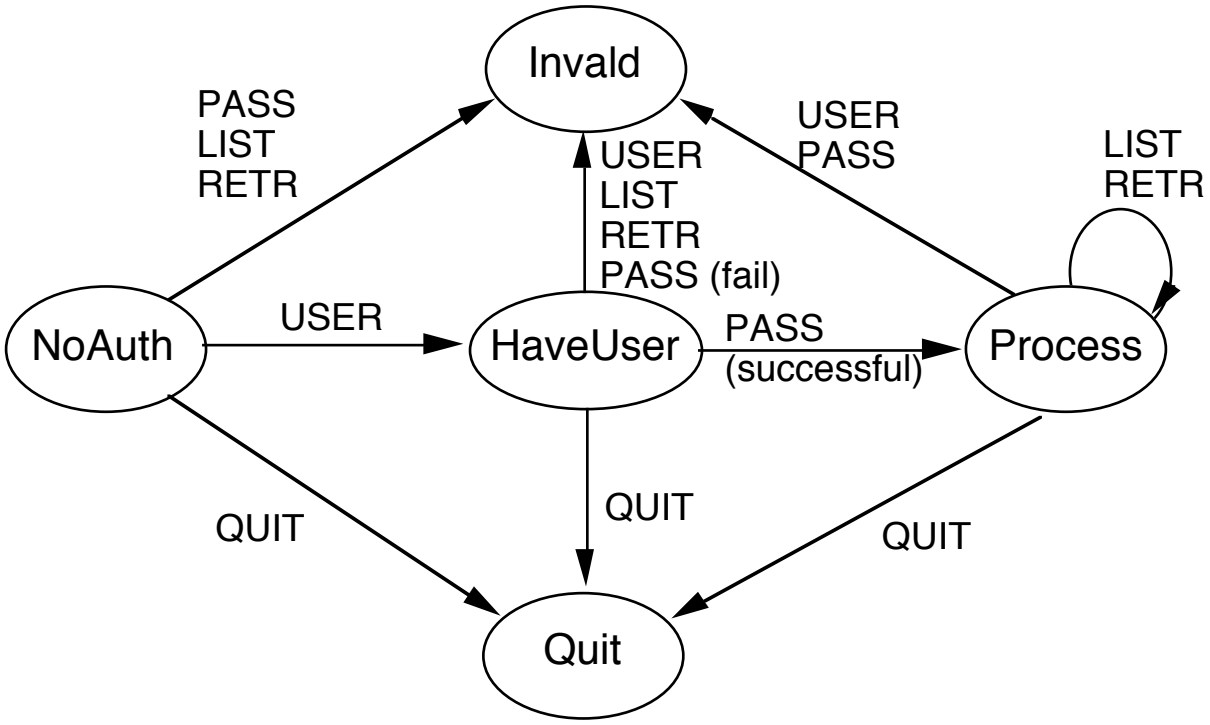
Some commands are only legal in some states

How to deal with states?

- If (case) statements
- Table of function pointers
- State Objects (State pattern)

Finite Automata - State Machines

A better way of looking at all of this is using a picture.



Naming the states

We will use the following names for the states:

0	NoAuth
1	HaveUser
2	Process
3	Invalid
4	Quit

Implementing a State Machine: switch

```
int state = 0;
while (true) {
   command = input.read();
   switch (state) {
      case 0:
         if (command.isUser()) {
            username = command.argument();
            state = 1;
         }
         else if (command.isQuit())
            state = 4;
         else
            error("Illegal command: " + command);
         break;
      case 1:
         if (command.isPassword()) {
            if (valid(username, command.argument()))
               state = 2;
            else {
               error("Unauthorized User");
               state = 3;
            }
         }
         else
            error("Unknown: " + command);
         break;
...
```

More Readable version

```
int state = 0;
while (true) {
   command = input.read();
   switch (state) {
      case NO_AUTH:
         noAuthorizationStateHandle( command );
         break;
      case HAVE_USER:
         haveUserStateHandle( command );
         break;
      case PROCESS:
         processStateHandle( command );
         break;
      case INVALID:
         invalidStateHandle( command );
         break;
      case QUIT:
         quitStateHandle( command );
         break;
   }
```

Example Continued

```
void noAuthorizationStateHandle(PopCommand a Command) {
   if (command.isUser()) {
      username = command.argument();
      state = 1;
   }
   else if (command.isQuit())
      state = 4;
   else
      error("Illegal command: " + command);
}
```

Switch Method Analysis

Disadvantages

- Hard to read.

  Need the state machine picture to understand what is going on.

- Hard to modify.

  If the protocol (and therefore the state machine) changes, the code will most likely have to be rewritten.

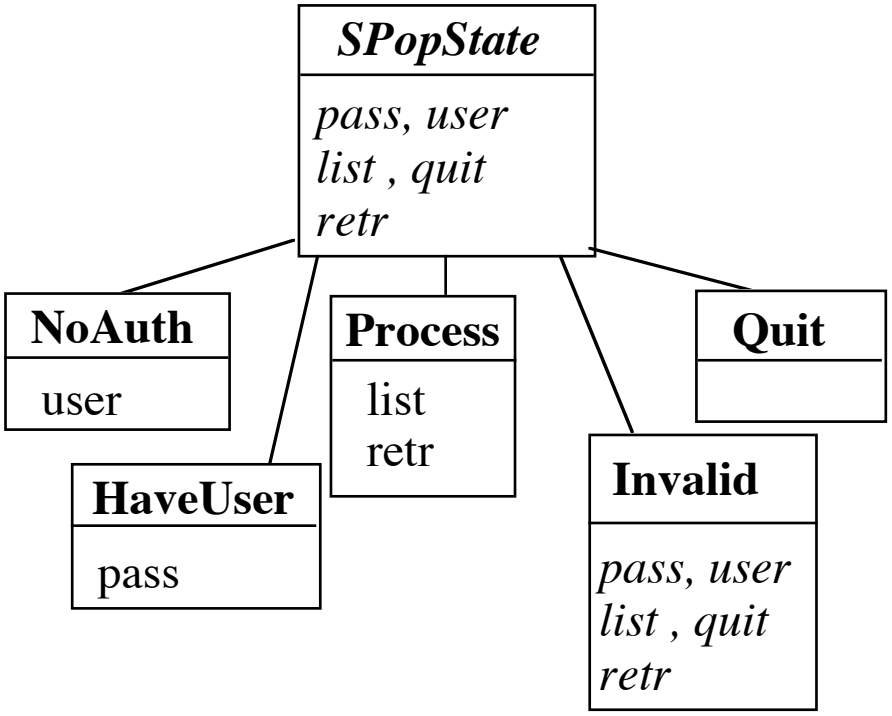
- Hard to debug.

- The code will get very long very quickly.

Advantages:

- The code within the while (true) can be put into a function and only called when there is new input.
- Everyone understands if statements
- Easy to start out with

Implementing a State Machine: Objects
The Basic Idea



Each method (pass, user, etc.) performs the proper action for the given state and returns the next state

SPopState is abstract state with the default behavior for each method

Server is done with client when we reach the Quit state, so I did not add methods to that state

Strawman Driver Program

```
class SPopServer
  {
  public void processRequest(InputStream in, OutputStream out,
    InetAddress clientAddress) throws IOException
    {

    SPopState currentState = new NoAuth();
    do
      {
      ProtocolParser requestData = new ProtocolParser( in );
      String request = requestData.getCommand();
      if ( request.isPassword() )
        currentState = currentState.pass( request, this);

      else if ( request.isUser())
        currentState = currentState.user(this);
      etc.

        send response to client
      }
    while ( ! currentState instanceof Quit  );
    }
  }
```

SPopState Implements Default Behavior

```
public class SPopState {
   public SPopState quit( SPopServer parent) {
      return new Quit();
      }

   public SPopState pass( PopCommand clientRequest, SPopServer parent)
      throws IllegalCommand {
      throw new IllegalCommand();
      }

   public SPopState user( PopCommand clientRequest, SPopServer parent)
      throws IllegalCommand {
      throw new IllegalCommand();
      }

   public SPopState list( PopCommand clientRequest, SPopServer parent)
      throws IllegalCommand {
      throw new IllegalCommand();
      }
```

Subclasses Implement Correct behavior for that State

```
public class NoAuth extends SPopState {
   public SPopState user( PopCommand clientRequest, SPopServer parent) {
      parent.setUser( clientRequest.getArgument() );
      parent.sendOKResponse();
      return new HaveUser();
   }
}

public class HaveUser extends SPopState {
   public SPopState pass( PopCommand clientRequest, SPopServer parent) {
      parent.setPassword( clientRequest.getArgument() );
      if ( parent.user&PasswordValid() ) {
         parent.sendOKResponse();
         return new Process();
      }
      else {
         parent.sendErrorResponse();
         return new NoAuth();
      }
   }
}
```

Smalltalk Example From VW 7 POP3 Client

Client has abstract state class Pop3State

Concrete states
- Pop3AuthorizationState
- Pop3TransactionState

Pop3Client that does the work

Pop3State

```
Smalltalk.Net defineClass: #Pop3State
   superclass: #{Core.Object}
   private: false
   instanceVariableNames: ''
   classInstanceVariableNames: ''
   category: 'Net-Pop Rocks'

Net.Pop3State methodsFor: 'commands'

delete: message for: connection
   Pop3StateError raiseSignal

list: number for: connection
   Pop3StateError raiseSignal

pass: aConnection
   Pop3StateError raiseSignal

quit: aConnection
   aConnection sendQuit

retrieveMessage: number for: connection
   Pop3StateError raiseSignal

stat: aConnection
   Pop3StateError raiseSignal

user: aConnection
   Pop3StateError raiseSignal
```

Pop3AuthorizationState

```
Smalltalk.Net defineClass: #Pop3AuthorizationState
   superclass: #{Net.Pop3State}
   indexedType: #none
   private: false
   instanceVariableNames: ''
   classInstanceVariableNames: ''
   imports: ''
   category: 'Net-Pop Rocks'

Net.Pop3AuthorizationState methodsFor: 'commands'

pass: aConnection
   aConnection sendPassword

user: aConnection
   aConnection sendUser
```

Pop3TransactionState

```
Smalltalk.Net defineClass: #Pop3TransactionState
   superclass: #{Net.Pop3State}
   indexedType: #none
   private: false
   instanceVariableNames: ''
   classInstanceVariableNames: ''
   imports: ''
   category: 'Net-Pop Rocks'


Net.Pop3TransactionState methodsFor: 'commands'

delete: message for: connection
   ^connection sendDeleteMessage: message

list: aConnection
   ^aConnection sendList

list: number for: connection
   ^connection sendList: number

retrieveMessage: number for: connection
   ^connection sendRetrieveMessage: number

stat: aConnection
   ^aConnection sendStat
```

Some Pop3Client Methods Using State

```
delete: message
   ^self state delete: message for: self

list
   ^self state list: self

list: messageNumber
   ^self state list: messageNumber for: self

quit
   self disconnect

retrieveMessage: number
   ^self state retrieveMessage: number for: self

status
   ^self state stat: self

login
   self state user: self.
   self hasPositiveResponse ifFalse: [^false].
   self state pass: self.
   self hasPositiveResponse ifFalse: [^false].
   self state: Pop3TransactionState new.
   ^true
```

Called By State

```
sendUser
   self sendCommand: ('USER <1s>' expandMacrosWith: self user username).
   self waitForResponse.
   self hasPositiveResponse
      ifFalse: [^NetClientError signalWith: #login message: self
serverResponse].

sendPassword
   self sendCommand: ('PASS <1s>' expandMacrosWith: self user password).
   self waitForResponse.
   self hasPositiveResponse
      ifFalse: [^NetClientError signalWith: #login message: self
serverResponse].
```

State Object Analysis

Problems

- Lots of little parts
- Algorithm distributed among different classes

Advantages:

- Easy to add new states
- Easy to change state transitions
- Each State class deals with one state

The Interesting Questions

Who does the actual work?

checking the password and user name

getting the mail messages

Who does the mapping from strings to functions?

How does all this fit together?

Does anyone understand this?

Special Bonus Question

Can you determine how to eliminate the need for mapping from strings to functions without using if statement (or switch statements)?

Implementing a State Machine with a Table

Commands	States				
	NoAuth	HaveUser	Process	Invalid	Quit
USER					
PASS					
LIST					
RETR					
QUIT					

Each cell needs:

- A function to process request
- Next state on success
- Next state on failure

The State Table

Commands	States				
	NoAuth	HaveUser	Process	Invalid	Quit
USER	actionUser	actionNull	actionNull		
	HaveUser	Invalid	Invalid	Quit	Quit
	Invalid	*Invalid*	*Invalid*	*Quit*	*Quit*
PASS	actionNull	actionPass	actionNull		
	Invalid	Process	Invalid	Quit	Quit
	Invalid	*Invalid*	*Invalid*	*Quit*	*Quit*
LIST	actionNull	actionNull	actionList		
	Invalid	Invalid	Process	Quit	Quit
	Invalid	*Invalid*	*Invalid?*	*Quit*	*Quit*
RETR	actionNull	actionNull	actionRetr		
	Invalid	Invalid	Process	Quit	Quit
	Invalid	*Invalid*	*Invalid?*	*Quit*	*Quit*
QUIT	actionQuit	actionQuit	actionQuit		
	Quit	Quit	Quit	Quit	Quit
	Quit	*Quit*	*Quit*	*Quit*	*Quit*

Key

Function to process request
Next State on success
Next State on failure

Basic Operation

Get request from user

Use current state and new request to find in table operation to perform

Perform the operation

Change state based on table and result of operation

How to place Operation in a Table

C/C++

- Use function pointers

Smalltalk

- Use symbols and reflection
- Use blocks

Java

- Use reflection
- Use Inner classes

Function Pointers in C/C++

```
void quickSort( int* array, int LowBound, int HighBound)
{
  // source code to sort array from LowBound to HighBound
  // using quicksort has been removed to save room on page
}

void mergeSort(int* array, int LowBound, int HighBound)
{ // same here}

void insertionSort(int* array, int LowBound, int HighBound)
{ // ditto }

void main()
{
  void (*sort) (int*, int, int);
  int size;
  int data[100];

  // pretend data and Size are initialized

  if (size < 25)
    sort = insertionSort;

  else if (size > 100)
    sort = quickSort;

  else
    sort = mergeSort;

  sort(data, 0, 99);
}
```

SPOP State table: C/C++

In C/C++ we can define the following:

```
struct
{
   int     currentState;
   char     *command;
   int     stateIfSucceed;
   int     stateIfFailed;
   int     (*action)(char **);
} actionTable[] =
{
   {0, "USER", 1, 3, actionUser},
   {0, "QUIT", 4, 4, actionQuit},
   {1, "PASS", 2, 3, actionPass},
   {1, "QUIT", 4, 4, actionQuit},
   {2, "LIST", 2, 2, actionList},
   {2, "RETR", 2, 2, actionList},
   {2, "QUIT", 4, 4, actionList},
   {0, 0, 0, 0, 0}
};
```

Advantages:

- Easy to see what is going on.
  Even easier if the states are given names.

- Easy to add new commands.

Smalltalk - Symbols & Reflection

Direct method execution

```
3 squared
2 + 3
'A cat in the hat' copyFrom: 3 to: 5
```

Method execution via reflection

```
3 perform: #squared
2 perform: #+ with: 3
'A cat in the hat' perform: #copyFrom:to: with: 3 with: 5
```

The method to execute is an argument of perform:

So store the symbol (#squared) of the method in the table

Function Pointers in Java
Use Reflection

Class.getMethod maps strings to methods

public Method getMethod(String name,Class parameterTypes[])
    throws NoSuchMethodException, SecurityException

Returns a Method object that reflects the specified public member method of the class or interface represented by this Class object. The name parameter is a String specifying the simple name the desired method, and the parameterTypes parameter is an array of Class objects that identify the method's formal parameter types, in declared order.

The method to reflect is located by searching all the member methods of the class or interface represented by this Class object for a public method with the specified name and exactly the same formal parameter types.

Throws: NoSuchMethodException
  if a matching method is not found.

Throws: SecurityException
  if access to the information is denied.

Simple Class for Example

```
class Example
  {
  public void getLunch()
    {
    System.out.println( "Lunch Time!");
    }

  public void getLunch( String day)
    {
    System.out.println( "Lunch Time for " + day);
    }

  public void eatOut( String where)
    {
    System.out.println( "MacDonalds? ");
    }

  public void eatOut( int where)
    {
    System.out.println( "PizzaHut? " + where );
    }
  }
```

Using Class.getMethod
Simple Example

```
import java.lang.reflect.Method;

class Test
  {
  public  static  void  main( String  args[] ) throws Exception
    {
    Example a = new Example();

    Class[] stringType = { Class.forName( "java.lang.String" ) };

    Object[] stringParameter = { "Monday" };

    Method tryMe;

    tryMe = a.getClass().getMethod( "getLunch", stringType );

    tryMe.invoke( a, stringParameter );

    }
  }
```

Output

```
Lunch Time for Monday
```

State Table Analysis

Advantages

- Compact view of states and transitions
- Easy to add remove states
- Easy to modify transitions

Disadvantages

- Language support varies
- Compile time checks are replaced by runtime check