CS 580 Client-Server Programming
Fall Semester, 2004
Doc 22 States

Contents

States .................................................. 2
Finite Automata - State Machines .......................... 3
Implementing a State Machine: switch .................. 4
Implementing a State Machine with a Table .......... 8
  Function Pointers in C/C++ .............................. 12
  Smalltalk - Symbols & Reflection ........................ 14
  Function Pointers in Java .............................. 15
Implementing a State Machine: Objects .................. 19
  Strawman Driver Program ............................... 20
  Smalltalk Example From VW 7 POP3 Client .......... 23

References

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

Past CS580 lecture notes

Copyright ©, All rights reserved. 2004 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.
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:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NoAuth</td>
</tr>
<tr>
<td>1</td>
<td>HaveUser</td>
</tr>
<tr>
<td>2</td>
<td>Process</td>
</tr>
<tr>
<td>3</td>
<td>Invalid</td>
</tr>
<tr>
<td>4</td>
<td>Quit</td>
</tr>
</tbody>
</table>
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

```java
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 with a Table

<table>
<thead>
<tr>
<th>Commands</th>
<th>States</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NoAuth</td>
</tr>
<tr>
<td>USER</td>
<td></td>
</tr>
<tr>
<td>PASS</td>
<td></td>
</tr>
<tr>
<td>LIST</td>
<td></td>
</tr>
<tr>
<td>RETR</td>
<td></td>
</tr>
<tr>
<td>QUIT</td>
<td></td>
</tr>
</tbody>
</table>

Each cell needs:

- A function to process request
- Next state on success
- Next state on failure
The State Table

<table>
<thead>
<tr>
<th>Commands</th>
<th>States</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NoAuth</td>
</tr>
<tr>
<td>USER</td>
<td>actionUser</td>
</tr>
<tr>
<td>HaveUser</td>
<td>Invalid</td>
</tr>
<tr>
<td>Invalid</td>
<td>Invalid</td>
</tr>
<tr>
<td>PASS</td>
<td>actionNull</td>
</tr>
<tr>
<td>Invalid</td>
<td>Process</td>
</tr>
<tr>
<td></td>
<td>Invalid</td>
</tr>
<tr>
<td>LIST</td>
<td>actionNull</td>
</tr>
<tr>
<td>Invalid</td>
<td>Invalid</td>
</tr>
<tr>
<td>Invalid</td>
<td>Invalid</td>
</tr>
<tr>
<td>RETR</td>
<td>actionNull</td>
</tr>
<tr>
<td>Invalid</td>
<td>Invalid</td>
</tr>
<tr>
<td>Invalid</td>
<td>Invalid</td>
</tr>
<tr>
<td>QUIT</td>
<td>actionQuit</td>
</tr>
<tr>
<td>Quit</td>
<td>Quit</td>
</tr>
<tr>
<td>Quit</td>
<td>Quit</td>
</tr>
</tbody>
</table>

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++

```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:

```c
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
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

```java
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
public class Pop3State {

    public Object listFor( int number, Pop3Client connection) {
        throw new Pop3StateError();
    }

    public void pass( Pop3Client aConnection) {
        throw new Pop3StateError();
    }

    public void quit( Pop3Client aConnection) {
        aConnection.sendQuit();
    }

    public Object retrieveMessageFor( int number, Pop3Client connection) {
        throw new Pop3StateError();
    }

    public Object stat( Pop3Client aConnection) {
        throw new Pop3StateError();
    }

    public void user( Pop3Client aConnection) {
        throw new Pop3StateError();
    }
}

Pop3State - Java Version
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
public class Pop3AuthorizationState extends Pop3State {

    public void pass( Pop3Client aConnection) {
        aConnection.sendPassword();
    }

    public void user( Pop3Client aConnection) {
        aConnection.sendUser();
    }
}
Pop3AuthorizationState – Smalltalk Version

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
public class Pop3TransactionState extends Pop3State {

    public Object deleteFor( message, Pop3Client connection) {
        return connection.sendDeleteMessage(message);
    }

    public Object list( Pop3Client aConnection) {
        return aConnection.sendList();
    }

    public Object listFor( int number, Pop3Client connection) {
        return connection.sendList( number);
    }

    public Object retrieveMessageFor( int number, Pop3Client connection) {
        return connection.sendRetrieveMessage( number);
    }

    public Object stat( Pop3Client aConnection) {
        return aConnection.sendStat();
    }

    public void user( Pop3Client aConnection) {
        throw new Pop3StateError();
    }
}
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
Pop3Client Declaration

Pop3Client is used by a GUI to interact with server.

public class Pop3Client extends NetClient {

    private NetUser user; //Contains user name and password
    private String hostName;
    private int portNumber;
    private int retries;
    private Socket connection;
    private ReadAppendStream stream;
    private Pop3State state;
    private String serverResponse;

    public Pop3Client() {
        this.state( new Pop3State());
    }

    public Pop3State state() {
        return state;
    }

    public void state(Pop3State aState) {
        state = aState;
    }

    many methods not shown
Login on – A Short trace of Action

(Pop3Client)
public boolean login() {
    this.state().user( this);
    if (this.hasPositiveResponse())
        return false;
    this.state().pass( this);
    if (this.hasPositiveResponse() )
        return false;
    this.state( new Pop3TransactionState());
    return true;
}

(Pop3AuthorizationState)
public void user( Pop3Client aConnection) {
    aConnection.sendUser();
}

(Pop3Client)
public void sendUser() {
    this.sendCommand("USER ", this.user().username());
    this.waitForResponse();
    if (!this.hasPositiveResponse() )
        throw new NetClientError();
}
public void sendCommand( aString) {
    this.log("C: ", hostname(), "S: ", aString);
    stream.write( aString , CRLF);
    stream.flush();
}

public void waitForResponse()
    String result = stream.throughAll( CRLF);
    this.serverResponse( result);
}

public boolean hasPositiveResponse() {
    return (this.serverResponse().startsWith("+OK"));
}

public void sendPassword() {
    this.sendCommand("PASS ", this.user().password());
    this.waitForResponse();
    if (!this.hasPositiveResponse() )
        throw new NetClientError();
}
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