

CS 580 Client-Server Programming
Spring Semester, 2006
Doc 17 States
Apr 4, 2006

Copyright ©, All rights reserved. 2006 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.

References

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

Selected Patterns for Implementing Finite State Machines, Paul Adamczyk,
http://hillside.net/plop/2004/papers/padamczyk0/PLoP2004_padamczyk0_0.doc,
referenced from http://hillside.net/plop/2004/final_submissions.html

States

Some Servers are stateful or have modes

Each connection has different states

Some commands are only legal in some states

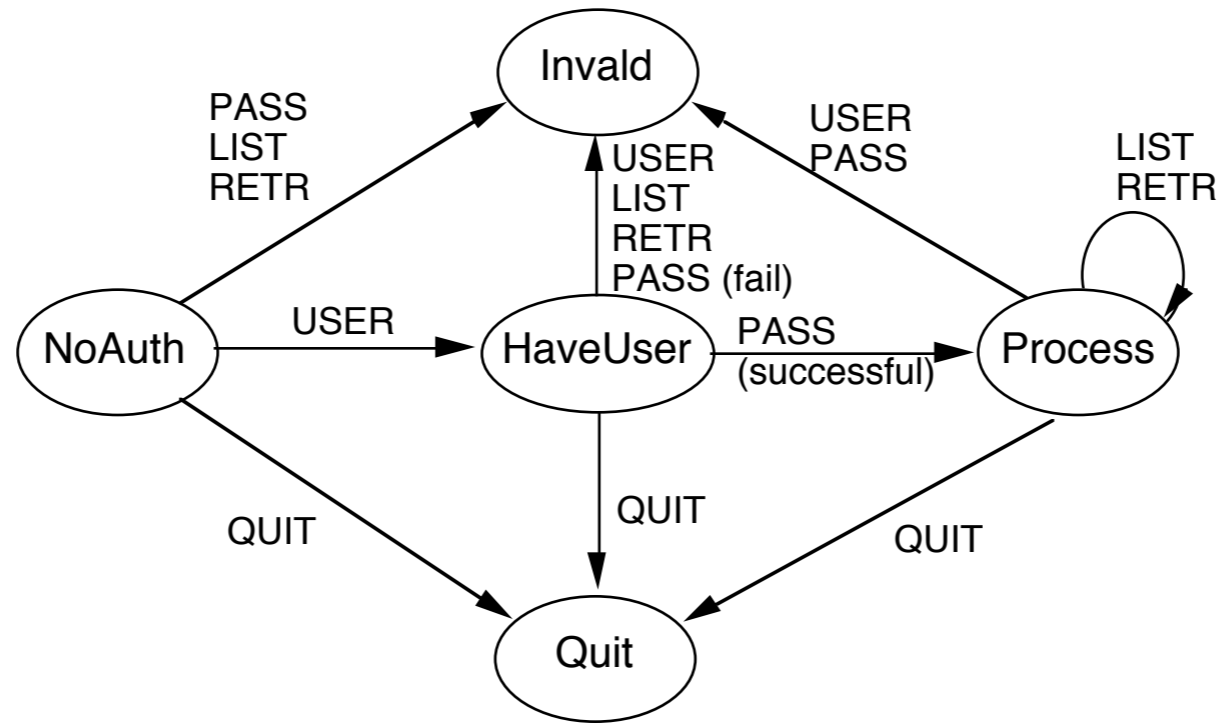
How to deal with states?

- If (case) statements

- Table of function pointers

- State Objects (State pattern)

Finite Automata - State Machines



Using Switch Statements

```
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;
    }
}
```

0	NoAuth
1	HaveUser
2	Process
3	Invalid
4	Quit

More Readable Version

```
int state = NO_AUTH;
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;
    }
}

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

Switch Method Analysis

Disadvantages

Hard to read for large or complex states

Hard to modify

Hard to debug

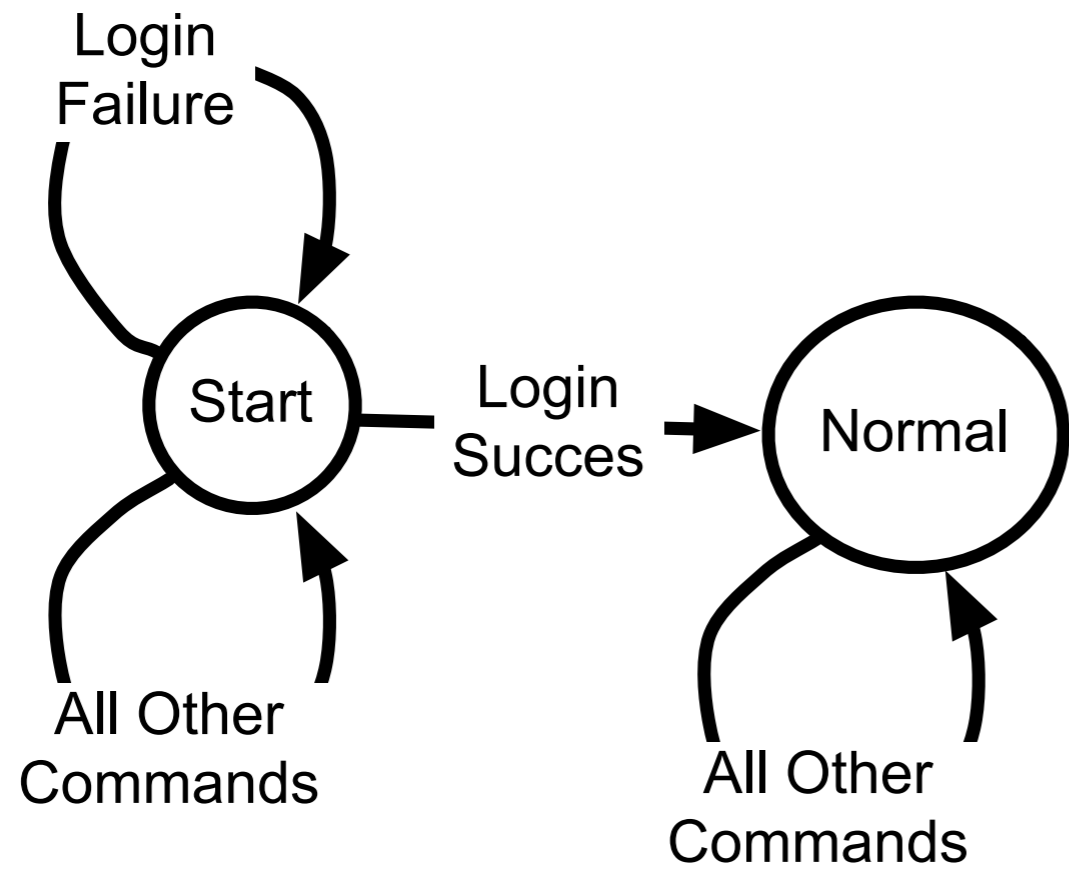
The code will get very long very quickly

Advantages

Everyone understands if statements

Simple for small/simple situations

Special Case



```
command = input.nextCommand()
```

```
if command.isLogin()
```

```
    process login
```

```
else
```

```
    handle illegal command
```

```
end
```

```
while !command.quit?
```

```
    command = input.nextCommand()
```

```
    process command
```

```
end
```


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

State Table Details

Commands	States				
	NoAuth	HaveUser	Process	Invalid	Quit
USER	actionUser	actionNull	actionNull		
	HaveUser	Invalid	Invalid	Quit	Quit
	<i>Invalid</i>	<i>Invalid</i>	<i>Invalid</i>	<i>Quit</i>	<i>Quit</i>
PASS	actionNull	actionPass	actionNull		
	Invalid	Process	Invalid	Quit	Quit
	<i>Invalid</i>	<i>Invalid</i>	<i>Invalid</i>	<i>Quit</i>	<i>Quit</i>
LIST	actionNull	actionNull	actionList		
	Invalid	Invalid	Process	Quit	Quit
	<i>Invalid</i>	<i>Invalid</i>	<i>Invalid?</i>	<i>Quit</i>	<i>Quit</i>
RETR	actionNull	actionNull	actionRetr		
	Invalid	Invalid	Process	Quit	Quit
	<i>Invalid</i>	<i>Invalid</i>	<i>Invalid?</i>	<i>Quit</i>	<i>Quit</i>
QUIT	actionQuit	actionQuit	actionQuit		
	Quit	Quit	Quit	Quit	Quit
	<i>Quit</i>	<i>Quit</i>	<i>Quit</i>	<i>Quit</i>	<i>Quit</i>

Function to process request
Next State on success
<i>Next State on failure</i>

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

Ruby

Use function references

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 in C/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}
};
```

0	NoAuth
1	HaveUser
2	Process
3	Invalid
4	Quit

Easy to see what is going on.

Easy to add new commands.

Ruby Method References

```
def cat()  
  puts 'dog'  
end
```

```
def increase(aNumber)  
  puts aNumber + 1  
end
```

```
x = method(:cat)  
x.call
```

```
y = method(:increase)  
y.call(4)
```

Ruby State Table

```
noAuth = {  
  #Command  Success  Fail State  action  
  :user => [:HaveUser, :Invalid, method(:actionUser)]  
  :quit => [:Quit, :Quit, method(:actionQuit)]  
  etc.  
}
```

```
def actionUser  
  blah  
end  
  
def actionQuit  
  blah  
end
```

```
haveUser = {  
  :pass => [:Process, :Invalid, method(:actionPass)]  
  :quit => [:Quit, :Quit, method(:actionQuit)]  
  etc  
}
```

```
stateTable = {  
  :NoAuth => noAuth  
  :HaveUser => haveUser  
  etc  
}
```

```
currentState = :NoAuth  
while currentState != :Quit  
  command = input.readCommmmand()  
  stateOperations = stateTable[currentState][command.symbol]  
  operationSucceeded? = stateOperations[3].call(command.data)  
  if operationSucceeded?  
    currentState = stateOperations[0]  
  else  
    currentState = stateOperations[1]  
  end  
end
```


State Table Analysis

Advantages

Compact view of states and transitions

Easy to add remove states

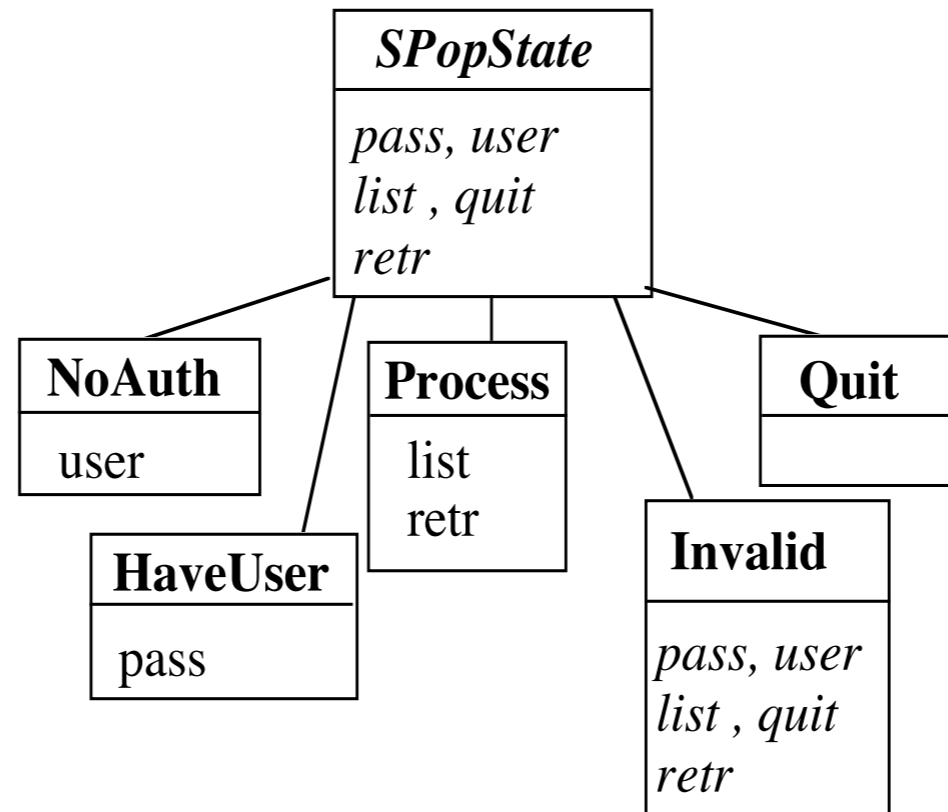
Easy to modify transitions

Disadvantages

Language support varies

Compile time checks are replaced by runtime check

Implementing a State Machine: Objects



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

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();
        }
    }
}
```

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