References

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

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

Diagram:
- **NoAuth**
  - USER
  - PASS
  - LIST
  - RETR
  - QUIT

- **HaveUser**
  - USER
  - PASS (fail)
  - PASS (successful)
  - QUIT

- **Invalid**
  - USER
  - PASS
  - LIST
  - RETR

- **Process**
  - LIST
  - RETR

- **Quit**
  - QUIT
Using Switch Statements

```java
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;
```
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
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

<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
## State Table Details

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

<table>
<thead>
<tr>
<th>Function to process request</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next State on success</td>
</tr>
<tr>
<td>Next State on failure</td>
</tr>
</tbody>
</table>
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++

```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++

```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}
};
```

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.
}

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.readCommand()
  stateOperations = stateTable[currentState][command.symbol]
  operationSucceeded? = stateOperations[3].call(command.data)
  if operationSucceeded?
    currentState = stateOperations[0]
  else
    currentState = stateOperations[1]
  end
end

def actionUser
  blah
end

def actionQuit
  blah
end
Java Reflection

Class.getMethod maps strings to method objects

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

Method.invoke() executes method objects

public Object invoke(Object receiver, Object... args)
A Class for an 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 );
    }
}
Reflection Example

```java
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 );
    }
}
```
Sample Table Entry

class StateTableEntry {
    int currentState;
    String command;
    int stateIfSucceed;
    int stateIfFailed;
    Method action;
}

StateTableEntry sample = new StateTableEntry();
Class[] stringType = { Class.forName( "java.lang.String" ) };
sample.action = Server.getMethod( "username", stringType );
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  );
    }
}
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();
        }
    }
}
```
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