References

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


Patterns of Enterprise Application Architecture, Martin Folwer, Addison-Wesley, 2003


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

- NoAuth
- HaveUser
- Process
- Quit

- PASS
- LIST
- RETR
- USER
- PASS
- (successful)
- PASS
- (fail)
- USER
- PASS
- LIST
- RETR
- QUIT
- 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
Special Case  Modularized

```java
processClientRequest(input, output) {
    try {
        boolean loginSucceeded = processLogin(input, output);
        if (!loginSucceeded) return;
        processAuthenticatedRequests(input, output);
    } catch (Exception error) {
        logger.throwing("ClientHandler","processingClientRequest", error);
    } finally {
        input.close();
        output.close();
    }
}
```
## 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>States</th>
<th>States</th>
<th>States</th>
<th>States</th>
<th>States</th>
<th>States</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NoAuth</td>
<td>HaveUser</td>
<td>Process</td>
<td>Invalid</td>
<td>Quit</td>
<td></td>
</tr>
<tr>
<td>USER</td>
<td>actionUser</td>
<td>actionNull</td>
<td>actionNull</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HaveUser</td>
<td>Invalid</td>
<td>Invalid</td>
<td>Quit</td>
<td>Quit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Invalid</td>
<td>Invalid</td>
<td>Invalid</td>
<td>Quit</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>
<td></td>
</tr>
<tr>
<td>Invalid</td>
<td>Process</td>
<td>Invalid</td>
<td>Quit</td>
<td>Quit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Invalid</td>
<td>Invalid</td>
<td>Invalid</td>
<td>Quit</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>
<td></td>
</tr>
<tr>
<td>Invalid</td>
<td>Invalid</td>
<td>Process</td>
<td>Quit</td>
<td>Quit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Invalid</td>
<td>Invalid</td>
<td>Invalid</td>
<td>Quit</td>
<td>Quit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RETR</td>
<td>actionNull</td>
<td>actionNull</td>
<td>actionRatr</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Invalid</td>
<td>Invalid</td>
<td>Process</td>
<td>Quit</td>
<td>Quit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Invalid</td>
<td>Invalid</td>
<td>Invalid</td>
<td>Quit</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>
<td></td>
</tr>
<tr>
<td>Quit</td>
<td>Quit</td>
<td>Quit</td>
<td>Quit</td>
<td>Quit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Quit</td>
<td>Quit</td>
<td>Quit</td>
<td>Quit</td>
<td>Quit</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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);
}
```

Sunday, December 2, 12
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.
Java Reflection

**Class.getMethod** maps strings to method objects

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

**Method.invoke()** executes method objects

```java
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

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
Database
Databases & Architecture

How to keep SQL isolated?

How to isolate database connection details?

How to keep dealing with the database under control?

How to structure programs that use databases?
Topics

Organizing Domain Logic

Structuring code that accesses database
Organizing Domain Logic

How to organize an application that uses a database

Fowler provides the following methods

Transaction Script
Domain Model
Table Module
Service Layer
Transaction Script

Each request from GUI or client can be viewed as a separate transaction

Each request/transaction is handled by a separate method

Consequences

Very simple to implement
As application grows in complexity, becomes overly complex and hard to manage
Domain Model

Implement classes that incorporates both behavior & data

Classes represent objects in the domain

Program becomes collection of interacting objects

Objects map to tables
  A single object may span many tables
  A table row may contain multiple objects

Consequences

Overly complex for simple applications
Scales well to complex applications
Database organizes data differently
Table Module

For each table (or view) implement a class

Each class holds the business logic related to the data in the table

Consequences

Classes are organized around database structure rather than OO principles

Handles more complex situations than Transaction Script

Not as scalable as Domain Model
Structuring code that accesses database

Hiding database connection details

Organizing Access to Database
Issues about Database Connections

Database usernames and passwords should not be scattered in code
How much database connection detail should be scattered in the code
public class DatabaseConnector {
    private String databaseUrl;
    private String user;
    private String password;
    private ArrayList connectionPool;

    private static DatabaseConnector instance =
        DatabaseConnector("filename");

    public static DatabaseConnector instance() {
        return instance;
    }

    private DatabaseConnector(String filename) {
        read file for database info
        set private fields
    }

    public ResultSet executeQuery( String sql ) {
        return getStatement().executeQuery( sql);
    }

    public Statement getStatement() {
        return getConnection().createStatement();
    }

    private Connection getConnection() { return a connection}
    etc
For Future Examples - Office Hours

Common Operations

Find Office hours for instructor X
Find office hours of any graduate advisor
Find office hours of any undergraduate advisor
Find office hours of any TA
Who has office hours at time X
What times are there no office hours
Add office hours
Modify office hours
### Faculty

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>Office</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Eckberg</td>
<td>GMCS-543</td>
<td>594-6834</td>
</tr>
<tr>
<td>2</td>
<td>Donald</td>
<td>GMCS-541</td>
<td>594-7248</td>
</tr>
<tr>
<td>3</td>
<td>Carroll</td>
<td>GMCS-537</td>
<td>594-7242</td>
</tr>
</tbody>
</table>

### RoleTypes

<table>
<thead>
<tr>
<th>ID</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Undergraduate Advisor</td>
</tr>
<tr>
<td>2</td>
<td>Graduate Advisor</td>
</tr>
<tr>
<td>3</td>
<td>TA</td>
</tr>
</tbody>
</table>

### OfficeHours

<table>
<thead>
<tr>
<th>Id</th>
<th>StartTime</th>
<th>EndTime</th>
<th>Day</th>
<th>FacultyId</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10:00</td>
<td>11:00</td>
<td>Tuesday</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>10:00</td>
<td>11:00</td>
<td>Thursday</td>
<td>1</td>
</tr>
</tbody>
</table>

### Roles

<table>
<thead>
<tr>
<th>FacultyId</th>
<th>Typeld</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>
Organizing Access to Database

Table Data Gateway
Row Data Gateway
Active Record
Data Mapper
Table Data Gateway

A wrapper for the SQL to access the table

One object handles all the rows in a table or view

Each table has one class that knows the table

One object represents the table – all the rows

Gateway hides all the Sql from the rest of the program

Works well with
  Table Module
  Transaction Script
public class FacultyTableGateway {
    public ResultSet findRow(int facultyId) {
        String select = "SELECT * FROM Faculty WHERE id = ?";
        Statement selectStatement = DatabaseConnector.instance().prepareStatement(select);
        selectStatement.setObject(1, facultyId);
        return selectStatement.executeQuery();
    }

    public ResultSet findAll() { code here }

    public ResultSet findWhere(String whereClause) { code here }

    public void update(int facultyId, String name, String office, String phone) { code here }

    public int insert(String name, String office, String phone) { add insert code here }

    public void delete(int facultyId) { code here }
}
What to return? Result sets or Objects?

Result set
   Handles multiple rows
   Tied to SQL

Objects
   How to deal with multiple rows
   Use
      Domain objects
      Collection
public class OfficeHoursServer {
    private OfficeHoursTableGateway officeHours;
    private FacultyTableGateway faculty;
    etc.

    public Vector officeHoursFor(String facultyName) {
        int facultyId = faculty.idFor(facultyName);
        ResultSet officeHoursRows = officeHours.officeHoursFor(facultyId);
        Vector officeHours = new Vector();
        while (officeHoursRows.next()) {
            Dictionary officeHour = new Dictionary();
            officeHour.put("start", officeHoursRows.getObject("start"));
            officeHour.put("end", officeHoursRows.getObject("end"));
            officeHour.put("day", officeHoursRows.getObject("day"));
            officeHours.add(officeHour);
        }
        officeHoursRows.close();
        return officeHours;
    }
}
Row Data Gateway

A Row Data Gateway gives you objects that look exactly like the record in your record structure but can be accessed with the regular mechanisms of your programming language.

One object handles or represents a single row in a table or view.

Each table has one class that knows the table.

Gateway hides all the SQL from the rest of the program.

A class provides just accessor methods to data in a row.

Works well with Transaction script.
public class FacultyRowGateway {
    private int id;
    private String name;
    private String office;
    private String phone;

    public void setName(String facultyName) {name = facultyName;}
    public void setOffice(String facultyOffice) {office = facultyOffice;}
    public void setPhone(String facultyPhone) {phone = facultyPhone;}
    public String getName() {return name;}
    public String getOffice() {return office;}
    public String getPhone() {return phone;
}
public int insert() {
    String insert = "INSERT INTO Faculty VALUES (?, ?, ?)";
    PreparedStatement insertStatement = DB.prepare(insert);
    try {
        insertStatement.setString(1,name);
        insertStatement.setString(2,office);
        insertStatement.setString(3,phone);
        insertStatement.execute();
        return getId();
    } catch (SQLException e) { handle exception}
}
FacultyRowGateway - Database Side

public void update() {code to update}

public void delete() {code to delete}

public static FacultyRowGateway find(int facultyId) {
    code to get data from database and create FacultyRowGateway
}

public static FacultyRowGateway find(String facultyName) {
    code to get data from database and create FacultyRowGateway
}
private static FacultyRowGateway load(ResultSet facultyData) {
    int id = facultyData.getInt(1);
    String name = facultyData.getString(2);
    String office = facultyData.getString(3);
    String phone = facultyData.getString(4);
    return new FacultyRowGateway(id, name, office, phone);
}

In Fowler's text he has a registry of faculty objects, so he first checks the registry for the faculty object before going to the database. Good idea, not done here just to keep the presentation simpler.
Using FacultyRowGateway

FacultyRowGateway newFaculty =
    new FacultyRowGateway("Pete", "GMCS 444", "594-2222");
newFaculty.insert();

FacultyRowGateway pete = FacultyRowGateway.find("Pete");
pete.setPhone("594-3333");
pete.update();
Active Record

Each domain object know how add/remove/find its state in the database

Class for each table
An object represents one row in the table
Similar to Row Data Gateway with domain logic
Faculty Example

```java
public class Faculty {
    String name;
    String phoneNumber;
    int id;
    etc.

    private final static String findByNameSql =
        "SELECT *
        FROM faculty
        WHERE name = '??';

    public static Faculty findByName(String name) {
        Statement find =
            databaseConnector.prepareStatement(findByNameSql);
        find.setObject(1, name);
        ResultSet facultyRow = find.executeQuery();
        return load(facultyRow);
    }

    private static Faculty load(ResultSet facultyRow) {
        create faculty object.
        get data out of ResultSet.
        Put data into faculty object.
    }
}```
Faculty Example

public boolean hasOfficeHoursAt(Time anHour) {
    Iterator hours = officeHours().iterator();
    while (hours.hasNext()) {
        OfficeHour officeHour = (OfficeHour) hours.next();
        if (officeHour.contains(anHour)) return true;
    }
    return false;
}

public ArrayList officeHours() {
    if (officeHours == nil) {
        officeHours = OfficeHour.findFor(id);
    }
    return officeHours;
}

etc.
Object-Relational Mapping Layers

Data Mapper

Implementing a good object-relational layer is a lot of work

Use existing tools to save a lot of time

Read/Write objects from tables without SQL

Some existing object-relational layers

- JDO – Java Data Object (Java framework)
- TopLink (Commercial - Java)
- Hibernate (Open source - Java)
- Cayenne (Open source - Java)
- GLORP (Open source - Smalltalk)
Hibernate Simple Example

Storing Person objects in table

Database Table

<table>
<thead>
<tr>
<th>id</th>
<th>first_name</th>
<th>last_name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SQL Used to Create Table

CREATE TABLE PEOPLE
    (FIRST_NAME varchar(50) NULL ,
     LAST_NAME varchar(50) NULL ,
     ID int NOT NULL ,
     PRIMARY KEY (id));
package sample;

public class Person {
    String firstName;
    String lastName;
    long id;

    public People () {super(); }

    public People(String first, String last) {
        firstName = first;
        lastName = last;
    }
    public String getLastName() { return lastName; }
    public String getFirstName() { return firstName; }
    public void setFirstName( String name) { firstName = name; }
    public void setLastName( String name) { lastName = name; }
    public long getId() { return id; }
    public void setId(long l) {id = l; }
    public String toString() {return firstName + " " + lastName + id; }
}
Mapping – Person.hbm.xml

Indicates how to map object fields to table columns

```xml
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
   "-//Hibernate/Hibernate Mapping DTD//EN"
   "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd" >

<hibernate-mapping package="sample">
   <class
      name="Person"
      table="people" >
      <id
         name="id"
         type="java.lang.Long"
         column="id" >
         <generator class="assigned"/>
      </id>
      <property
         name="firstName"
         column="first_name"
         type="string"
         not-null="false"
         length="50" />
      <property
         name="lastName"
         column="last_name"
         type="string"
         not-null="false"
         length="50" />
   </class>
</hibernate-mapping>
```
public class Main {
    public static void main(String[] args) throws Exception {
        sampleRead();
        sampleWrite();
    }

    static Session getHibernateSession() throws MappingException, HibernateException, Exception {
        some code to get HibernateSession
    }

    static void sampleWrite() throws MappingException, HibernateException, Exception {
        Session session = getHibernateSession();
        Transaction save = session.beginTransaction();
        Person newPerson = new Person("Jack", "Frost");
        newPerson.setId(1);
        session.save(newPerson);
        newPerson = new Person("Jack", "Ripper");
        newPerson.setId(2);
        session.save(newPerson);
        save.commit();
        session.close();
    }
}

Sample Connection

Sunday, December 2, 12
static void sampleRead() throws MappingException, HibernateException, Exception {
    Session session = getHibernateSession();
    Query getByLastName =
        session.createQuery(
            "from People p where p.lastName = :var"");
    getByLastName.setString("var", "Frost");
    List result = getByLastName.list();
    System.out.println("Number of Objects: " + result.size());
    Person frost = (Person) result.get(0);
    System.out.println(frost);
    session.close();
}
O-R Mapping - Vietnam of Computer Science


http://www.codinghorror.com/blog/archives/000621.html

Last mile problem & OR mapping Problem

Dual-Schema Problem
Entity Identity Issues
The Data Retrieval Mechanism
  Query-By-Example (QBE)
  Query-By-API (QBA)
  Query-By-Language (QBL)
Some Solutions

Abandon relational databases - store objects
Abandon objects
Abandon OR-layers
Accept OR-Layer limitations
  Use SQL when easier
Add relational concepts to language
Add relational concepts to frameworks