MIS Quarterly

RESEARCH ARTICLE

DISTRIBUTED COGNITION IN SOFTWARE DESIGN: AN EXPERIMENTAL INVESTIGATION OF THE ROLE OF DESIGN PATTERNS AND COLLABORATION

George Mangalaraj
College of Business and Technology, Western Illinois University, Macomb, IL 61455 U.S.A. {g-mangalaraj@wiu.edu}

Sridhar Nerur, RadhaKanta Mahapatra, and Kenneth H. Price
College of Business Administration, University of Texas at Arlington, Arlington, TX 76019-0437 U.S.A.
{snerur@uta.edu} {mahapatra@uta.edu} {price@uta.edu}

Appendix A

An Illustrative Example of the Application of a Design Pattern

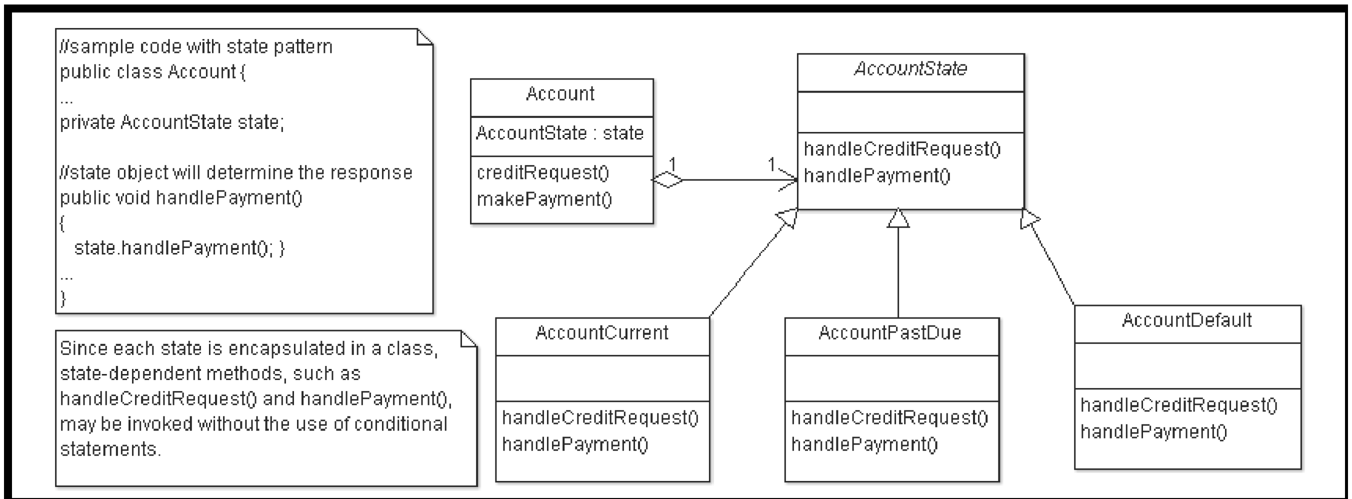
The state pattern allows an object to exhibit a different behavior (i.e., alter its response to one or more messages) when its internal state changes (Gamma et al. 1995). Consider an Account class that keeps track of a customer's credit payments. At any time during its life cycle, such an Account object can be in one of many possible states: Current, Canceled, and PastDue, to name but a few. The state of the Account object would determine its response to requests/messages that it receives from other objects. For example, a request to handle payment (i.e., invocation of the handlePayment() method) would result in different behaviors depending on whether it is in the Current, Canceled, or PastDue state. When the state pattern is not used, a variable (e.g., a String) would typically be used to maintain the state of the object. As shown below, the use of such a String variable (let us call it *status*) would result in conditional statements in the handle payment operation.

```
public void handlePayment() {
    if ( status.equals("Current"))
        // statements to handle payment when the Account object is current
    else if ( status.equals("Canceled"))
        //statements to handle payment when the Account object is in the canceled state
    else
        //statements to handle payment when the Account object is past due
}
```

Clearly, the method would have to be changed every time a new state is introduced. For example, if we decide to introduce a state called Hold, the condition for hold would have to be included in the if…then statements.

The state pattern advocates maintaining a state object rather than a variable such as status, so that the Account object's behavior can be changed easily at run-time while giving us the flexibility to add states in the future without modifications to the state-dependent methods themselves. Therefore, in our example, we would introduce an abstract class called AccountState that presents an interface common to its subclasses, Current, Canceled, and PastDue (see Figure A1). Note that there are as many subclasses as there are conditional branches shown in the code above. The Account object now maintains a reference to a state object (i.e., an object of type AccountState), indicating its present state. All requests (such as handlePayment()) that are dependent on the state would be delegated to this state object. Therefore, when the state pattern is applied, the snippet of code would be as follows:

```
public void handlePayment() {
        state.handlePayment(); //where state is an AccountState object
}
```



```
//sample code with state pattern
public class Account {
...
private AccountState state;

//state object will determine the response
public void handlePayment()
{
  state.handlePayment(); }
...
}
```

Since each state is encapsulated in a class, state-dependent methods, such as handleCreditRequest() and handlePayment(), may be invoked without the use of conditional statements.

Figure A1. Current, Cancelled, and Past Due Subclasses

Reference

Gamma, E., Helm, R., Johnson, R. E., and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software,* Reading, MA: Addison-Wesley.

Appendix B

Questionnaire

1. Please circle your gender: Male Female

2. Please indicate your age on your last birthday __________

3. Highest educational level:

a) High school	b) Technical school or community college	c) Undergraduate degree
d) Graduate degree	d) Doctoral degree	e) Other ____________

4. Indicate number of years of your programming experience in any programming language?

   a) 0 – 1   b) 1 – 2   c) 2 – 4   d) 4 – 6   e) > 6

5. Indicate number of years of your programming experience in object-oriented languages?

   a) 0 – 1   b) 1 – 2   c) 2 – 4   d) 4 – 6   e) > 6

6. What would you consider to be your level of experience in object-oriented design?

   a) No experience   b) Novice   c) Intermediate   d) Expert

7. What would you consider to be your level of experience in design patterns?

   a) No experience   b) Novice   c) Intermediate   d) Expert

8. What object-oriented programming languages are you familiar with?

a) C++	b) C#	c) Java	d) Small Talk	e) Objective-C
f) Eiffel	g) Python	h) VB.NET	i) Others ____________	

9. Before today's task performance, have you ever worked with your partner?*

   a) Yes   b) No

10. How do you feel about your overall experience of working on the task today?

Very Dissatisfied	1	2	3	4	5	6	7	Very Satisfied
Very Displeased	1	2	3	4	5	6	7	Very Pleased
Very Frustrated	1	2	3	4	5	6	7	Very Contented
Absolutely Terrible	1	2	3	4	5	6	7	Absolutely Delighted

11. My partner actively participated in solving the design problem*

Strongly Disagree	1	2	3	4	5	6	7	Strongly Agree

12. What patterns did you use to arrive at the solution?

Note: *These questions were asked only in the collaborating pair condition.

Appendix C

The Two Experimental Tasks

Warm-Up Task: Duck Pond Simulation Game[1]

XYZ Corporation is in the process of designing a duck pond simulation game. The game can display a large variety of duck species swimming and making quacking sounds. Some of the duck types that are planned to be used in the simulation game include: Mallard Duck, Redhead Duck, Bufflehead Duck, and Pintail Duck. They are also planning to add Rubber Duck to the game. Real ducks are capable of quacking and flying whereas the rubber ducks can only squeak and cannot fly. All the ducks have a behavior to display themselves on the screen.
You are required to draw a class diagram that can be used to implement the system described above.

Main Task: Weather Monitor[2]

You must develop software that allows clients to periodically check for changes in the status of sensors in a weather monitoring station. Examples of various sensors could be temperature gauge, pressure gauge, humidity sensor, etc. A client must be able to create a monitor that will periodically check a particular sensor in the network. If the state of that sensor has changed since the monitor last checked the sensor, the monitor should write an event to a global event log.

A sensor is uniquely designated by its sensor ID. When making its initial monitoring request, the client specifies the ID of the sensor to be monitored and the monitoring period. At that point, the monitor is initialized but has not yet been started. The client makes a subsequent request to start the monitor. The client should be able to start and stop the monitor at any time.

Each sensor has an interface to check its current state, although the precise interface differs from sensor to sensor. As an example, to check a temperature sensor, you invoke its getTemparature method, whereas to check a relative humidity sensor, you call its getRh method. The various sensor classes are provided by different vendors, so you are not permitted to change the interfaces of those classes.

The components that make up the states of different sensors may also differ. For example, the state of a temperature sensor is defined by a floating point value. A relative humidity sensor state, on the other hand, is represented by a string.

Assume the existence of an Event Log and an Event class. The Event Log classes define a method, logEvent that takes an Event as an argument and places that Event in the Log. You should write a specific type of Event, a Sensor Change Event that includes the ID of the sensor.

You are required to draw a class diagram that can be used to implement the system described above.

[1]This problem description is adapted from the one found in E. Freeman, E.Freeman, B. Bates, and K. Sierra, *Head First Design Patterns*, Sebastopol, CA: O'Reilly Media, 2004.

[2]This problem description is loosely based on an example described in Freeman et al. (p. 56) and incorporates additional design constraints based on design insights from C. Richter, "Design Problems and Object-Oriented Solutions (http://www.oeng.com/problemsandsolutions.htm; retrieved February 10, 2013).

Appendix D

Grading Sheet for Weather Monitoring Station Problem

Subject ID:

No.	Description	Purpose	Points	Points Scored	Remarks
A. Classes					
1	Monitor	Sensors are monitored by this class	5		
2	Sensor	Can be abstract or an interface	5		
3	Specific Sensor	Temperature, Pressure sensors etc.	5		
4	Event log	Log of various events	5		
5	Event	Generic class	5		
6	Change Event	This is the sensor change event	5		
7	Adapter class	Adapter for the sensors to handle different states	10		
8	PollToken	To store state and compare	10		
B. Associations					
1	Monitor/Sensors Interface	Polltoken object is returned to monitor	5		
2	Event logging	Monitor detects changes when they occur, creates change event	10		
3	Specific sensor/ Poll token	Specific sensors creating poll token	5		
4	Comparing states	Monitor should be able to compare poll tokens (previous state poll token is the current state poll token)	10		
6	Sensor Adapter/ Sensor	Sensor Adapters forward poll to specific sensors	10		
C. Optional Classes					
1					
2					
3					
D. Unnecessary/Wrong Classes					
1			-		
2			-		
3			-		
	Total				

Patterns Used:

Note: Class names are given for illustration only and the evaluated solution can have different names for them.