IMPLEMENTING DATABASE SOLUTIONS ON THE WEB WITH ASPECT-ORIENTED PROGRAMMING

John Stamey (jwstamey@coastal.edu) & Bryan Saunders (btsaunde@coastal.edu)
Department of Computer Science, Coastal Carolina University, Conway, SC 29528

ABSTRACT

This paper uses Aspect-Oriented Programming to provide an elegant solution to the problem of concurrent database access on the web. Code tangling and code scattering, typical problems in development of such Java-based solutions, disappear during the implementation of concurrency (avoiding the lost update, the dirty read, and the inconsistent analysis) and serializability (using two-phase locking protocol).

INTRODUCTION

Most all robust web applications provide Add, Browse, Change, and Delete capabilities. While such applications are straightforward to implement in a single-user environment, the multi-user platform of the web poses a number of additional design and coding challenges. Primary concerns are that concurrency and locking are not explicitly provided by database management systems such as Oracle, MySQL, Postrgres, or SQLserver at the web-application level.

We use Aspect-Oriented Programming (AOP) to provide an elegant solution to the problem of database implementations on the web that provide multi-user access to information. The skill level needed to implement such solutions has been found to be appropriate for advanced undergraduate Computer Science majors at Coastal Carolina University. We present a discussion of database concurrency and locking problems for web-based applications, and report a development solution using Java with AOP.

SOLVING PROGRAMMING PROBLEMS WITH ASPECTS

Internet database connectivity, implemented with Java servlets, provides an excellent platform to see many benefits of Aspect-Oriented programming. While most modern database management systems provide concurrency control in “console” or administrator mode, Internet delivery of databases requires that programmers implement concurrency control through the application code. One important issue is avoidance of concurrency problems such as the lost update, the uncommitted dependency (dirty read), and the inconsistent analysis (nonrepeatable read) [1]. A second issue is providing for serialization of transactions implemented through two-phase locking protocols (basic, conservative, strict, and rigorous). [3, 8]

An *aspect* is a fundamental unit in an object-oriented programming language, much the same as a class. Aspects contain data, methods, and objects of other classes, just like classes do. Aspects are instantiated at runtime, based on program execution, exhibiting polymorphic behavior. System development can benefit from the use of aspects to perform some of the modularization of code. This is most effective during the initial development when cold blocks (or methods) are called from multiple points. Aspects are also valuable in helping maintain modularity through changes and additional

releases [6]. As system modifications and new releases occur, the idea of streamlining that code (basically, refactoring [5]) can be overlooked or suffer from a low position on the priority list. Resulting problems code scattering and code tangling then plague the system. It is entirely appropriate, and even necessary, that techniques to overcome such problems be addressed at the undergraduate level.

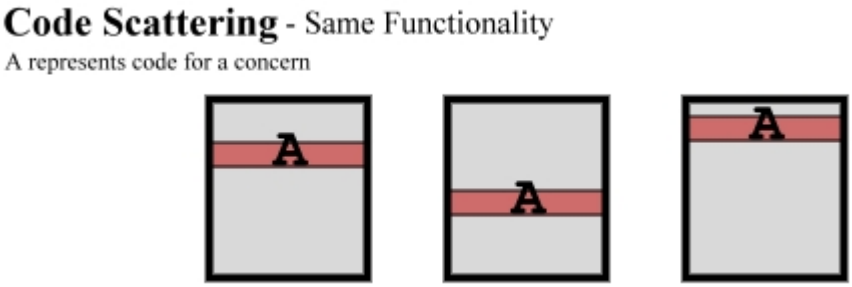

Figure 1

A *concern* is a system requirement. Well-known concerns include security, logging, authentication and authorization, and business rules. Each concerns executes from code locations called a *joinpoint.* A *pointcut* is a collection of joinpoints for one specific concern. *Crosscutting concerns* occur when a particular system requirement/concern is implemented in more than one location [4]. An effective strategy for implementing of concerns is at the heart of AOP. We will now examine two common problems that occur when concerns and crosscutting concerns are not effectively managed: code scattering and code tangling.

One type of *code scattering* occurs when concerns are implemented in more than one location. Implementing similar logic in more than one location is a problem called code scattering (Figure 1). Such coding practices can increase the time necessary to modify code, and lessen the chance of reusing modules containing specialized code. A second type of code scattering occurs when related tasks have multiple implementations. Related tasks could reasonably be implemented in the same module (Figure 2).

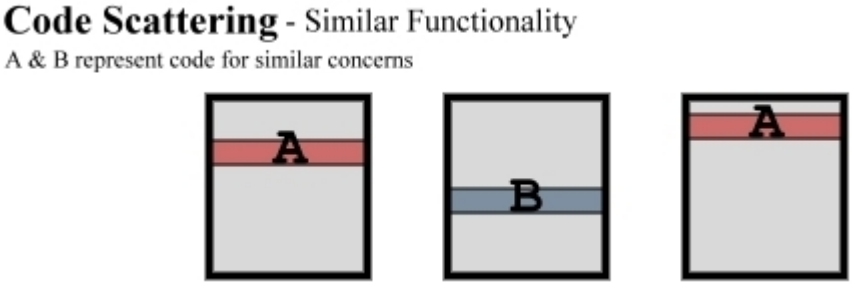

Figure 2

Good design practices can avoid code scattering when a program is first written. Programs, however, are guilty of evolution more often than not. Instead of adding operations with redundant code to address concerns, careful analysis should lead to the creation of an aspect to address concerns in multiple locations. Code scattering causes a

number of problems for future development and releases, including increased the time required to modify or enhance code, and a lower level of reusability of modules.

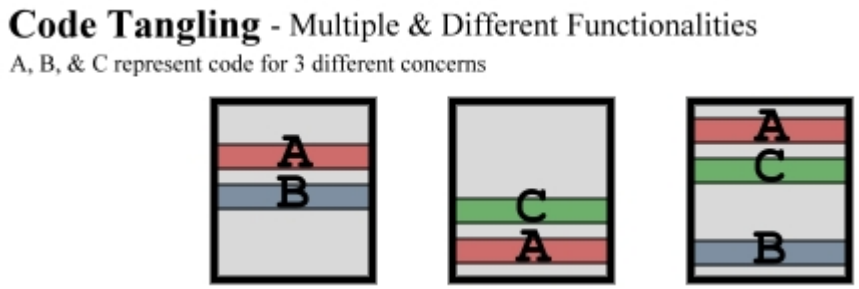

Figure 3

A different problem occurs when multiple tasks/concerns are implemented within the same programming logic. This problem of *code tangling* can lead to unnecessarily confusing implementations of code across a system (Figure 3).

A DATABASE/CONCURRENCY PROGRAMMING EXAMPLE

The following specifications were developed to provide an introductory-level scenario through which basic two-phase locking provided serializable transactions, along with establishing control of concurrency:

- Each user who logs into the system is assigned a unique session ID.
- Read or update/write transactions set the requesting user as the owner of the lock.
- When a read or update transaction has been completed (and the user returns to the main menu), the lock is released if that user is the owner of the lock. That is always the case with a write/exclusive lock. It is only the case when the user is the owner of the read/shared lock.

The following table provides some detail about transaction and lock states:

ACTION	LOCK	CONSEQUENCE
Read	0	Lock Set to 1, Data Retrieved, No message
Read	1	Lock Unchanged, Data Retrieved, No message
Read	2	Lock Unchanged, Data in Database Retrieved, Warning Message "Data May Change"
Write	0	Lock Set to 2, Data Retrieved anticipating change
Write	1	Lock Set to 2, Data Retrieved anticipating change
Write	2	Lock Unchanged, Warning Message "Data Is Updated Try Again Later"

The AOP solution to this problem was implemented through the Jakarta Tomcat server (http://jakarta.apache.org/tomcat/), running on Windows XP. The Eclipse Java compiler (http://www.eclipse.org/) was used with the AspectJ extension to Java (http://www.eclipse.org/aspectj). Implementing serializable transactions through a basic

two-phase locking protocol detailed in Elmasre and Navathe [2, p. 664] was used in this programming example. Assuming X is a tuple in the database:

1. All transactions issue readLock(X) or writeLock(X) before any readItem(X) is performed.
2. A transaction must issue a writeLock(X) before any writeItem(X) is performed.
3. A transaction must issue the operation unlock(X) after all readItem(X) and writeItem(X) operations are completed in the transaction.
4. A transaction will not issue a readLock(X) operation if it already holds a read (shared) lock or a write (exclusive) lock on item X.
5. A transaction will not issue a writeLock(X) operation if it already holds a read/shared lock or a write/exclusive lock on item X. (In the example, one cannot upgrade a lock from read/shared to write/exclusive; to write, one must first release the read lock).
6. A transaction will not issue an unlock(X) operation unless it already holds a read/shared or write/exclusive lock on X.

All of these points are identified on the code found in Appendix A.

IMPLEMENTING ASPECTS

Aspects are implemented in units called *advice*; advice is a construct, much like a function or method, consisting of code that is to be executed at a joinpoint. *After Advice* contains code that executes following a joinpoint. *Before Advice* contains code that executes before a joinpoint. *Around Advice* executes before a joinpoint and can actually alter the execution or data of the code following the joinpoint (or within a method, if the joinpoint is, itself, a method) [7]. The advice code written for this system is found in Appendix A. The Around Advice in Example 1, executed prior to processing each of the servlets Add, Browse, Change and Delete, freezes a row to check the status of the lock. Based on the status of the lock and the operation chosen, the user is allowed to proceed with the transaction (read or update), or the transaction is halted.

```
// Example 1 – Line 7 in Appendix A
void around(HttpServletRequest rq,HttpServletResponse rs):
call(void *.moveToProcess(HttpServletRequest,HttpServletResponse))
&& args(rq,rs)
```

The After Advice found in Examples 2 and 3 compare the sessionID and lock level against the lock table, to check for the event that a lock should be released once the process has finished.

```
// Example 2
after(HttpServletRequest rq,HttpServletResponse rs): call(void
doPost(HttpServletRequest,HttpServletResponse)) && args(rq,rs)

// Example 3
after(HttpSession session): call(void doLocks(HttpSession))
&& args(session)
```

Code scattering could occur if aspects were not used to check and update the locks, as each of the four available operations (Add, Browse, Change, and Delete) require a concurrency control check before and after they are executed, as well as a check of the row lock. Aspects, therefore, provide a straightforward way to combine locking and concurrency control into one operation. For the size of this application, it is reasonable to combine checks for locking and concurrency control in the same advice. However, if the application were to grow in size and scope, this would be a perfect opportunity to split these functionalities apart into separate aspects in order to avoid code tangling.

SUMMARY AND CONCLUSIONS

A useful programming problem has been presented and solved using AOP. Aspects solve several problems that occur during both refactoring and system enhancement. Lessons learned from this project include:

- Good initial design can modularize concerns without aspects, where a design is created during a first development iteration
- As code is changed, such as the development of upgrades or new code releases, concerns can become duplicated throughout a system. Aspects are a powerful tool to alleviate this problem of redundancy (code scattering and code tangling) as systems develop and mature in the following ways:
    - Code scattering can be avoided by recognizing instances of duplicate or complementary implementations of concerns and replacing this code with aspects
    - Code tangling can be avoided by recognizing multiple concerns with duplicate implementations and replacing this code with aspects

The code for this paper may be found at www.softwareengineeringonline.com/Aspects

REFERENCES

[1] Date, C.J. *An Introduction to Database Systems, 7th Edition.* Addison-Wesley Longman, 2001.

[2] Elmasri, R. & Navathe, S.B. *Fundamentals of Database Systems, 4th Edition*. Addison Wesley, 2003.

[3] Eswaran, K.P., Gray, J.N., Lorie, R.A. & Traiger, I.L. *The notions of consistency and predicate locks in a database system.* Communications of the ACM 19, 11 (Nov. 1976), 624-633.

[4] Elrad, T., Filman, T.E. & Bader, A., Editors, *Communications of the ACM*, Vol. 44., No. 10, October 2001, pp. 29-97 (Collected Articles)

[5] Fowler, M. Found at http://www.refactoring.com/.

[6] Kiczales, G. Aspect-Oriented Programming. ACM Computing Surveys, December 1996.

[7] Laddad, R. *AspectJ in Action.* Manning (2003).

[8] Stearns, R., Lewis, P.M. & Rosenkrantz, D.J. *Concurrency control for database systems.* In Proceedings of the 12th IEEE Symposium on Foundations of Computer Science (Oct. 1976). IEEE, New York, 1976, pp. 19-32.

APPENDIX A.

```
import java.sql.*;import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public aspect Aspect {
private static boolean locksEnabled = true;
private DBConn con = new DBConn();
void around(HttpServletRequest rq,HttpServletResponse rs): call(void
*.moveToProcess(HttpServletRequest,HttpServletResponse)) && args(rq,rs){
   HttpSession session = rq.getSession();
   String temp = String.valueOf(session.getAttribute("curOp"));
   char op = Character.toUpperCase(temp.charAt(0));
   int row = Integer.parseInt(String.valueOf(session.getAttribute("curRow")));
   int lock; con.Connect();
   if(locksEnabled){ try{ switch(op){
      case 'A': // There is never any locking on an Add, so we proceed()
         con.Disconnect(); proceed(rq,rs); break;
      case 'B': // If lock <= 1 - Unwarned Browsing - If lock >= 2 - Warned Browsing
         con.freeze(session,row); // Freeze Row
         lock = con.getLock(row); // Get Lock State – Point 1 (Line 37)
         if(lock <= 1){
            // Unwarned Browse Allowed
            con.setLock(session,row,1); // Set Lock to 1 – Point 1 (Line 40)
            con.deFreeze(row); // DeFreeze Row
            con.Disconnect();Main.redirect("Browse",rs);
         } else {
            // Warned Browse Allowed
            session.setAttribute("curMsg","<font color=#990000>Data May Change!</font>");
            con.deFreeze(row); // DeFreeze Row
            con.Disconnect();Main.redirect("Browse",rs);
         }  break;
      case 'C': // If lock <= 1 - Allow Change - If lock >= 2 - Force Warned Browsing
         con.freeze(session,row); // Freeze Row
         lock = con.getLock(row); // Get Lock State
         if(lock <= 1){ // Change Allowed
            con.setLock(session,row,2); // Set Lock to 2 – Point 2 (Line 59)
            con.deFreeze(row); // DeFreeze Row
            con.Disconnect();Main.redirect("Change",rs);
         } else { // Warned Browse Allowed
            session.setAttribute("curMsg","<font color=#990000>Data May Change!</font>");
            con.deFreeze(row); // DeFreeze Row
            con.Disconnect();Main.redirect("Browse",rs);
         }  break;
      default: proceed(rq,rs);
   }  }catch(IOException e){
      // Caught an Exception. So we shall proceed.
      System.err.println("Aspect Exception: "+e);
      e.printStackTrace(); session.setAttribute("curOp","M"); proceed(rq,rs);
   }  }else{
      // Locks are Disabled, so we proceed()
      con.Disconnect(); proceed(rq,rs); } }//End Advice
/* Start Point 6 */ after(HttpServletRequest rq,HttpServletResponse rs): call(void
doPost(HttpServletRequest,HttpServletResponse)) && args(rq,rs){
   HttpSession session = rq.getSession(); con.Connect();
   // Release Locks
   String sID = session.getId();
   int row = Integer.parseInt(String.valueOf(session.getAttribute("curRow")));
   if(con.isLockOwner(sID,row)){
      // We need to Release our Locks
      con.unLock(row);  } // Point 3 (Line 97)
   con.Disconnect();
}//End Advice
after(HttpSession session): call(void doLocks(HttpSession)) && args(session){
   con.Connect(); // Release Locks
   String sID = session.getId();
   int row = Integer.parseInt(String.valueOf(session.getAttribute("curRow")));
   if(con.isLockOwner(sID,row)){ // We need to Release our Locks
      con.unLock(row);
   } con.Disconnect(); }//End Advice }//End Aspect
```