AOSD 2006

5th International Conference on
Aspect-Oriented Software Development

Bonn, Germany
March 20-24, 2006

Industry Track Proceedings

Matt Chapman
Alexandre Vasseur
Günter Kniesel
(Eds.)

Technical Report IAI-TR-2006-3
ISSN 0944-8535

Institut für Informatik III
Universität Bonn

universitätbonn

Fifth International Conference on Aspect-Oriented Software Development

Bonn, Germany
March 20-24, 2006

Industry Track

Improving the Performance of Database Applications with Aspect-Oriented Programming

Uwe Hohenstein
Siemens AG
CT SE 2
D-81730 Muenchen
GERMANY
+49 89 636 44011
Uwe.Hohenstein@siemens.com

ABSTRACT

The performance of relational database applications often suffers. The reason is that query optimizers require accurate statistics about data in the database in order to provide optimal query execution plans. Unfortunately, the computation of these statistics must be initiated explicitly (e.g., within application code), and computing statistics takes some time. Moreover, it is not easy to decide when to update statistics of what tables in an application. A well-engineered solution requires adding source code usually in many places of an application.

The issue of updating the statistics for database optimization is a crosscutting concern. Thus we propose to use aspect-orientation to automate the calculation. We show how nicely the update functionality can be modularized in an aspect and how easy it is to specify the exact places and the time when statistics updates should be performed to speed up complex queries. Due to the automatic nature, computation takes place on time for complex queries, only when necessary, and only for stale tables.

The implementation language for the automated aspect-oriented statistics update concern is AspectJ, a well known and mature aspect-oriented programming language. The approach can however be implemented in any other aspect-oriented language. Unlike in traditional object-oriented pattern solutions, e.g. using the interceptor pattern, we do not have to modify existing code.

Categories and Subject Descriptors

H3.4 [**Information Storage and Retrieval**]: Systems and Software. *Performance evaluation (efficiency and effectiveness).*

General Terms

Management, Measurement, Performance, Languages.

Keywords

Database Optimization, Crosscutting Concerns, Aspect-Oriented Programming.

1. INTRODUCTION

A database system is nearly almost the most important factor for the performance of database applications. Particularly, relational database applications often suffer from a bad performance. Relational database management systems (RDBMSs) land themselves in it. The query and manipulation language SQL is designed to allow for an easy access to databases, it provides powerful logic-oriented query capabilities. In fact, researchers [1,2,3] and RDBMS vendors spent a lot of effort on query optimization [8] to find efficient execution plans. Hence, users are encouraged to formulate complex SQL statements, mostly joining several tables and filtering records using conditions, in order to exploit the full power of SQL. Internally, a DBMS's query optimizer has to find a high-performing execution plan for each query.

DBMS vendors recommend using cost-based query optimization (CBO) instead of rule-based optimization because CBO is the more recent technology and provides better execution plans. Moreover, newer concepts such as star queries [10,11] are only optimized with CBO. Besides existing indexes, CBO takes into account the amount of data in tables and indexes as well as the selectivity of attributes, i.e., how many records match a certain attribute value [8].

Users are often surprised that even with powerful CBO technology the performance is sometimes insufficient. Thereby, it is often overlooked that CBO needs precise database statistics about the amount of data in tables, the selectivity of attributes etc. [2]. These data statistics are not yet managed by the DBMS automatically, but the calculation can be requested by the user, explicitly on demand.

Supplying up-to-date database statistics is not as easy as it seems to be. We are faced with two problems.

- The calculation is time-consuming for larger tables, since it uses the complete data of a table (and associated indexes) ignoring previously calculated statistics. It is still up to recent research to find solutions for an incremental estimation [10]. Certainly, statistics can be *estimated* [14,15] instead of computing them completely to reduce the effort, but often this provides bad results.
- It is not easy to decide *when* to perform the calculation on *which* tables. If the calculation is initiated on several tables right before an expensive access operation, this might even slow the operation down. In general, a periodic calculation or an update before an application

starts is not appropriate because data is changing heavily and frequently.

We were confronted with these problems in a project in the telecommunication domain using an Oracle database. On the one hand, a huge amount of complex queries took several minutes (due to stale statistics), but could have been reduced to milliseconds if the statistics would have been up-to-date; thus having up-to-date database statistics was strongly required. On the other hand it took between 15 minutes up to 60 minutes to update the statistics, thus doing this just in time was not possible either. Using estimated statistics, several complex queries were still badly optimized.

The performance problems are astonishing because the total amount of data is not really huge. Furthermore, it should be mentioned that the database design was appropriate.

A sophisticated approach is required to achieve good query performance. The idea is

- to trigger the computation of the database statistics in the application code, whenever accurate statistics are necessary,
- but only for those tables that are used in queries and that have changed dramatically since the last computation; this limits the calculation time.

In this paper, we describe how to achieve this goal by means of aspect-orientation (AO). AO aims at providing systematic means for effective modularization of crosscutting concerns avoiding scattering of code [6]. Concerns such as logging and tracing are often described as classical candidates for using AO. Recent research has also shown usefulness in several other areas: [16] uses AO to separate concurrency control and failure handling code in a distributed system. The book of Rashid [22] gives an excellent overview about ongoing research in the context of databases and discusses all facets of AO: AO to implement DBMSs in a more modularized manner, persistence for aspects, and ideas on a persistence framework [21].

We here take benefit from AO to extend existing code in a non-invasive and modular manner. Before presenting the details of our AO approach, we discuss the basic requirements for an adequate environment in Section 2.

Section 3 is then presenting a flexible environment for updating statistics on demand. In spite of being useful, the approach is not easy to handle, unfortunately. The main problems are discussed in Section 4, particularly misusage, detection of stale tables etc.

The AO solution that solves these problems is presented in Section 5. We use AspectJ [15], a general-purpose aspect-oriented extension of Java, to gather statistical data for an effective cost-based optimization. In order to demonstrate the approach in detail, we here focus on Java programs using JDBC, but it is no problem to transfer the ideas to C++ and AspectC++.

2. BASIC REQUIREMENTS

The major problem is that calculating statistics on the one hand can last several minutes, but is necessary for complex queries on the other hand. An adequate approach must take into account at least the following points:

1. The approach should provide up-to-date database statistics for applications only when needed (in order not to delay an application), and on time for complex query execution.
2. It should check for *stale* tables (for which the statistics are not up-to-date) in order to reduce the time by computing only the statistics of relevant tables.
3. The environment should be central in the sense that an application will benefit from database statistics that are already updated by another application on the same tables.
4. The environment itself must not put much additional load on performance. This is essentially a matter of implementing the approach efficiently, but also a matter of using the environment correctly.
5. The mechanism should be stable against misusage. The performance behavior should not suffer even if the environment is not used adequately.

3. A FLEXIBLE ENVIRONMENT

We start with an intuitive environment that has been proposed by some developers. Application programmers can use the environment to control gathering database statistics. Technically speaking, the proposed environment essentially consists of the following class `CBO_Mgr`:

```
public class CBO_Mgr
{  public void Notify(List tables);
   public boolean Request(List tables,
                          boolean blocking);
   private static void Performer();
   private static List RequestList;
}
```

The *Notify* method let applications indicate larger modifications of data. An application should invoke this method whenever a lot of data has changed, e.g., if huge amounts of data have been inserted, modified, or deleted; the affected tables are passed in the *tables* parameter. Determining the list is left to the applications and has to be identified statically. These tables are subject to statistics calculation in future, but the updating of statistics does not occur immediately. When *Notify* is called, the list of tables is only dumped into the *RequestList* data member.

Applications can request the environment to update database statistics. The *Request* method should be called by an application whenever it performs time-consuming SELECT, DELETE or UPDATE queries with complex search conditions that require the database statistics to be up-to-date for certain tables. The relevant tables are passed in the *tables* parameter. The environment decides for which tables to calculate the statistics. The method first checks

whether the statistics for the passed tables are stale by checking the *RequestList* member. If there are no stale tables, it immediately returns "true" to the caller, because the statistics are up-to-date.

If there are stale statistics for the passed tables, the behavior depends on the *blocking* parameter. In case of a non-blocking execution, *Request* calls a *Performer* asynchronously, and returns false. The *Performer* then calculates the statistics of stale tables while the application is continuing in parallel. If the parameter asks for blocking execution, *Request* handles the requested tables synchronously, i.e., it waits for the termination of *Performer* and then returns true to the caller. In both cases, the tables are removed from the *RequestList*.

Without any explicit request, the statistics are additionally updated periodically. That is, the *Performer* method is started periodically by means of a *background job*. This handles the union of all notified tables, evaluates the database statistics for them, and then deletes the handled tables from the *RequestList*. Thus, we will get correct database statistics in certain time intervals, the frequency of which must be fixed appropriately.

The purpose of *Notify* is to indicate what tables have been changed. Unfortunately (and surprisingly), it is not possible to obtain this information from a RDBMS. A general possibility is to compare the real number of records in a table (`SELECT COUNT(*)`) with the number of records the DBMS has noticed (in the system views). But such a check is too expensive for the purpose of checking staleness (cf. Requirement 4). Since we cannot rely on an easy and fast way to get all the stale tables, we have to take *Notify* as the source for staleness.

This *Notify* information is then used by *Request* to reduce the amount of tables for collecting database statistics and to avoid useless requests on already up-to-date tables. We remember analyzing the statistics of a lot of tables takes several minutes. When the environment is correctly used, then it is enough to update the statistics of only those tables that are asked for (by *Request*) and are also notified (by *Notify*). This is a small and reasonable set of tables.

The non-blocking mode of *Request* is useful to update the statistics of relevant tables as early as possible, hopefully being finished on time. The application is not delayed by waiting for up-to-date statistics. *Request* in blocking mode computes database statistics immediately and lets applications wait for completion. This is useful whenever we must be sure that database statistics are up-to-date for certain tables.

Computing database statistics periodically by a background job is useful to handle the tables mentioned in *Notify*, but for which no explicit request was received recently, maybe forgotten.

There should be a central process managing the *RequestList*. This process will avoid repeated updates of database statistics among applications. All applications are then able to share the same environment and could benefit from global statistics updates. It does not matter who initiates the calculation.

4. PROBLEMS

The above approach seems to be intuitive and reasonable. Unfortunately there are some problems to avoid useless *Request* invocations. As a general point, we should keep in mind any kind of misuse (see Section 2, requirements 4 and 5). There is a strong danger of using the environment incorrectly. For example, programmers can forget *Notify* and/or *Request*, use both in wrong places, pass wrong tables (irrelevant ones), or miss relevant ones. These points will endanger the performance.

If a *Notify* about larger table modifications is missing, the environment gets the impression that there are no stale tables. Hence, no calculation of statistics will take place.

If a *Request* is missing, then complex queries will run longer, in case larger modifications have taken place. If performance problems have been detected and localized, the places where *Requests* are missing can at least be found.

If a *Notify* is wrongly called for a table, e.g., a table that has not changed much, then the table is considered stale. This presumably causes an unnecessary calculation of statistics for that table. This can lead to a delay since the computation of statistics takes place without any need. The same happens for *Requests* on wrong tables.

It is not easy to insert *Notify* and *Request* into existing code appropriately. In fact, it is a manual task for an application developer to decide where to put them. For sure, the analysis must be done carefully in order to avoid a bad performance behavior.

A static analysis alone is not enough, the control flow must be considered, too. To use the environment, we must know what parts of the code, particularly what SQL statements, are really executed, what loops are executed how often, what IF and what ELSE cases are relevant. For example, if we place *Notify* and *Request* on the same tables within a loop, then the computation of statistics takes place for every iteration. Then statistics calculation is not only useless, but leads to a performance penalty.

Even if we know that an SQL statement is performed, e.g., an UPDATE, we do not know how many records are affected. Should *Notify* be called or is it not necessary because only a few records are affected? That is, we should take into account the amount of changed data. The SQL statement can certainly be asked, but this information must then be actively evaluated. Even if only one record is updated, this could be done in a loop that is executed a thousand times. Then it is worth to notify the environment about the table change after the loop.

It is very difficult to find a pragmatic solution that is easy to use, and that can be brought into existing applications without any danger of misuse or performance degradation.

5. AUTOMATIC APPROACH

As we have seen, a manual approach must be handled carefully – in spite of having an adequate environment. Particularly, problems with robustness in the presence of misuse must be avoided. Anyway, a deeper code analysis is absolutely indispensable and a lot of code has to be touched manually.

We now show another approach taking benefit from aspect-orientation programming (AOP), more precisely the AspectJ language, to solve these major problems.

5.1 Aspect-Orientation

The evolution of software development techniques has been driven by the need to achieve a better separation of crosscutting concerns (CCCs). CCCs are those functionalities that are typically spread across several classes. They often lead to lower programming productivity, poor quality and traceability, lower degree of code reuse, and the lack of supporting evolution [17]. Patterns [7] can help to deal with such CCCs, but provide only partial solutions and do not capture the concerns explicitly.

AOP introduces new concepts to capture CCCs in modules called aspects. The language AspectJ [15], as one popular representative of an AOP language, is an aspect-oriented extension to Java. The language extensions support the modularization of crosscutting concerns to avoid code tangling and code scattering. Most important for our purpose is that AspectJ allows us to add behaviour to *existing* code without touching the original code at hundreds of places.

Programming with AspectJ is essentially done by modularizing basic concerns in ordinary objects (as in object-orientation) and crosscutting concerns in *aspects*. Aspects are special units that crosscut the objects and define some crosscutting functionality.

The main purpose of aspects is to change the dynamic structure of a program by modifying the program flow. An aspect can intercept certain points of the program flow, called *join points,* and introduce aspect code there. To this end, an aspect declares *pointcuts* that specify sets of join points by means of a signature expression. Pointcuts determine what join points should be trapped in the program flow. Wildcards can be used to select several methods of several classes easily. For instance, `MyClass*.get*(..)` selects all methods that start with "get" in all classes with a name starting with the string "MyClass". Parameter types can be fixed or left open with `(..)`. Pointcuts can also be combined by usual logical operators `!`, `&&` and `||`.

Once join points are captured, *advices* specify weaving rules involving those joint points, such as taking a certain action before, after or instead of the join points. In addition, pointcuts can be specified in such a way that they expose the context at the matched join point, i.e., the caller, the callee and parameter values are accessible by the advice.

5.2 Updating Database Statistics with AspectJ

We here assume that applications are written in Java using the standard interface JDBC (Java Database Connectivity) for database accesses. Then we can benefit from the aspect-oriented language AspectJ that extends Java. However, C++ programs with database accesses in ODBC, embedded SQL, or any Microsoft database interface can be handled the same way by using AspectC++, which is similar to AspectJ in concepts, but extends C++.

The general principle is as follows: AspectJ helps us to trap any execution of relevant JDBC statements by specifying the right pointcuts. Moreover, it allows us to extract the SQL statements to be performed and to get the number of modified records. Thanks to AspectJ, it is possible to detect any major changes in tables, to insert not only a *Notify* after INSERT, DELETE and UPDATE statements, but also to notify only if a certain threshold of modified records is passed.

Similarly, AspectJ let us detect any SELECT, DELETE and UPDATE with a complex WHERE-part; a blocking *Request* can be added in front of them to enforce up-to-date statistics. The advantages of using AspectJ are obvious:

- There is no manual activity of inserting *Notify* and *Request*; any misusage is excluded. The number of changes to any table is correctly recorded, and statistics are thus only computed if reasonable: We can rely upon.
- The coding can easily be done outside the existing application code by means of aspects: Aspects implicitly work on all the code.

Let us now dive into the technical details. Figure 1 defines an aspect `CBO_KEEPER`. This aspect specifies the JDBC join points in the code by means of `pointcut`s. Each join point causes execution of an advice, i.e., additional logic for our CBO environment. `CBO_KEEPER` also introduces a hashmap `changeCounters` in order to maintain a record counter for each table.

The pointcuts define the signature of relevant invocation of JDBC statements. `jdbcExecuteQuery` specifies any immediate invocation of a query in JDBC, e.g., `stmt.executeQuery("SELECT ...")`. The SQL query itself is passed as a string. The pointcut `call(* java.sql.Statement+.executeQuery(String))` catches any such call of `executeQuery` on an object of class `Statement` in package `java.sql` with a `String` parameter, i.e., the query to be performed. '`Statement+`' denotes that the objects can be of class `Statement` itself or of any subclass. Since we need information about the passed query string, we have to bind a variable `str` to the parameter value by means of `args(str)`. `str` can be used in advices later on.

```
public aspect CBO_KEEPER
{
  HashMap changeCounters = new HashMap(); // <tabName,cnt> manages counter for each table
  public pointcut jdbcExecuteQuery (String str) :                 // direct query
    call(* java.sql.Statement+.executeQuery(String)) && args(str);
  public pointcut jdbcExecuteUpdate (String str) :                // direct update
    call(* java.sql.Statement+.executeUpdate(String)) && args(str);
  public pointcut jdbcExecutePrepare(PreparedStatement s):        // execute prepared stmt
    (call(* java.sql.PreparedStatement.executeUpdate()) ||
     call(* java.sql.PreparedStatement.executeQuery())    ) && target(stmt);
  // advices for pointcuts:
  after(PreparedStatement stmt) returning (int cnt) : jdbcExecutePrepare(stmt)
                                          // cnt == number of modified records
  { String theTable;
    int newCnt = 0;
    check type of SQL statement by parsing stmt.toString() and determine table => theTable
    Integer c = (Integer)changeCounter.get(theTable);
    if (c != null)
         newCnt = c.intValue() + cnt;
    else newCnt = cnt;
    changeCounter.put(theTable, new Integer(newCnt));
  }
  before(String str):jdbcExecuteQuery(str) || jdbcExecuteUpdate(str)
  { determine type and relevant tables from str;
    ArrayList tabList = new ArrayList;
    for each table occurring in query
    { Integer cnt = (Integer)changeCounter.get(table);
      int c = 0;
      if (cnt != null) {
           c = cnt.intValue();
      if (c>=5000 && condition=COMPLEX) { // threshold passed
         tabList.insert(table);
    }
    ComputeStatistics(tabList); // blocking
  }
}
```

Figure 1: Aspect CBO_Keeper

Similarly, `jdbcExecuteUpdate` traps any immediate invocation of a manipulation (`UPDATE`, `DELETE`, `INSERT`), i.e., invocations of the form

```
cnt = stmt.executeUpdate("UPDATE ...").
```

In order to speed up applications, so-called prepared statements are used in JDBC:

```
PreparedStatement pstmt =
       conn.prepareStatement("UPDATE ...");
pstmt.setInt(1,9);
pstmt.setString(2,"abc");
      // set two parameters for 1st execution
cnt = pstmt.executeUpdate(); // 1st execution
     // set two parameters for 2nd execution
pstmt.setInt(1,5);
pstmt.setString(2,"def");
cnt = pstmt.executeUpdate(); // 2nd execution
…
```

`prepareStatement` prepares a parameterized SQL statement. The statement is analyzed only once by the DBMS and executed several times for different parameter values (9, "abc") and (5,"def") by means of `executeUpdate()`.

We could define a pointcut for statements of the form `pstmt = conn.prepareStatement("UPDATE ...")` in order to obtain SQL operations to be performed. But the execution is later done by `pstmt.executeQuery()` or `pstmt.executeUpdate()` after having set parameter values. Hence, we would have to pass the SQL statement to another pointcut trapping the `executeUpdate`. However, it is easier to trap the execution of a prepared statement: We track all executions of `pstmt.executeQuery/Update()` by means of `jdbcExecutePrepare`. Here we need access to the `this` object achieved by `target(stmt)` binding the `PreparedStatement` to `stmt`. Then, `stmt.toString()` can be used to determine the SQL statement.

Advices in aspect `CBO_KEEPER` will now use these joinpoints and parameter information to handle *Notify* and *Request* adequately at the joinpoints.

The clause `after() : jdbcExecutePrepare(stmt)` is used to add code after an invocation of the join points specified by the pointcut `jdbcExecutePrepare`, i.e., after having executed a statement. The `returning` clause binds a variable `cnt` to the value returned by `stmt.executeUpdate()`, i.e., we can access the number of affected records by `cnt`. The executed statement is accessible by passing the `PreparedStatement` parameter `stmt` to the pointcut and advice. The advice can obtain the SQL string by `stmt.toString()` and can extract the affected table. It then either inserts a new entry into `changeCounters` (if a table is not existing) or increments the value by `cnt`, the number of affected records.

Direct executions of statements can be handled by defining an analogous advice for `jdbcExecuteUpdate`.

Similarly, a `before` advice for `jdbcExecuteQuery/Update` checks for executed SQL queries (SELECT, UPDATE, DELETE) whether they are complex. The calculation of statistics can then be forced if necessary.

The query to be performed is accessible as SQL string `str`. The string can be parsed to determine whether the query is considered to be complex, for instance, if several tables are joined or if the query contains GROUP-BY-HAVING. In addition, the advice extracts the table name(s) from `str`. For each affected table, the `changeCounter` is asked for the number of collected modifications. If a certain threshold is passed (here 5000), then the computation of statistics is done for the table, and the counter is reset for the table. Thus, the calculation is only performed if reasonable. Both after and before advices now implement the Notify/Request mechanism.

Using the same principle, we can also collect other useful information, e.g., to keep track of the current number of records in tables without performing `SELECT COUNT(*)`. This information is indeed available in system views of the DBMS, but only if the statistics are up-to-date. The number of records in a table can be taken as another indicator for complex queries. In principle, we can also replace JDBC code by means of an `around` advice, particularly SQL strings, e.g., adding optimizer hints to SQL statements.

It is important to note that we have now reliable and quantified information about staleness: The number of modified records is determined automatically, while a manual *Notify* could have been forgotten in a manual approach. But some decisions have still to be taken: the thresholds and the question what queries are complex.

One additional point is important here. So far, any DELETE, UPDATE and INSERT statement is trapped, no matter whether the surrounding transaction is committed or rolled back. Since only committed transactions have to be taken into account, we catch any rollback and reset the `changeCounter` table to the values at the beginning of the transaction by means of

```
public pointcut jdbcRollback :
  call(void java.sql.Connection.rollback());
before jdbcRollback() {
  resetChangeCounter(); }
```

The above presentation calculates statistics in blocking mode, immediately before complex queries are executed. It would be nice to take benefit from an asynchronous statistics calculation. A non-blocking *Request* could prepare the calculation of statistics as early as possible, e.g., right after a *Notify*, whereas the blocking *Request* is invoked before complex queries are executed. The latter is just to be sure that the statistics are really up-to-date and to wait for completion if necessary. Unfortunately, it is not possible to push back a non-blocking *Request* to the place where large modifications take place. The power of AspectJ is not sufficient to this end. However, we can make a workaround: We define a second higher threshold, and if this threshold is passed, then the statistics calculation is performed in non-blocking mode in addition to gathering changes.

5.3 Results

We tried to assess the effectiveness of our AO approach. A precise performance analysis is not easy to perform. We can certainly compare our solution with the original application, the performance of which is bad and highly non-deterministic. This is useless.

At a first glance, it seems to be reasonable to compete with a perfect, highly optimized system. But the effort will be too high to achieve a perfect performance for existing applications. e.g., by adding optimizer hints – in fact, that is why we were looking for simpler alternatives. We can also obtain a perfect performance for the existing system by updating statistics permanently for any complex query inside the applications (however, without measuring the times for calculating statistics). This is again very difficult to achieve. Moreover, such a test would run days.

What we did was to compare our solution with one that calculates the database statistics only at the beginning of each application. Well, this will not give a perfect performance as an application is also changing data before complex queries run – but performance comparisons are never fair anyway.

We then used a simple definition of complex query implemented by the parser: A query is complex if either two large tables (> 10,000 records), more than three tables, or a GROUP BY are involved. We tested several threshold values in our environment to fine-tune the configuration of parameters. Finally, the presented approach achieves a performance improvement of 25%. This result is remarkable because we count the times for statistics calculation only in our proposal, but we do not for the original application. Hence, the delay we introduced owing to non-blocking *Request*s is more than compensated by an improved performance. This underlines the necessity for having nearly accurate statistics since queries, executed in

loop hundreds of times, took minutes otherwise. In general, we can state that the overhead produced by AspectJ is low, i.e., the code inserted by AspectJ into Java for maintaining table counters etc.

5.4 Alternatives

Since people were skeptic about using a new technology such as AspectJ, performing "strange" byte-code weaving, we were forced to look for alternative solutions. In fact, well-known design patterns such as Interceptor or Decorator can be used. However, they require some manual preparation for plugging in additional behavior. We neglected such an approach because several hundreds lines of code would have to be touched, selected very carefully.

A similar environment to what has been proposed could use database triggers. Triggers are able to catch modifications on database tables and to call other database operations, e.g., inserting or updating information in another table. Here, AFTER INSERT/UPDATE/DELETE ROW triggers can be defined for each relevant table; statement triggers cannot be used because it is not possible to compute the number of affected records within the trigger. Indeed, row triggers can count records in local variables, and statement triggers then increment the counters, however, now using a table `COUNTER(tabname, cnt)` in the database.

In contrast to the AspectJ solution, the performance degradation is much higher, since each inserted, updated, and deleted record is accompanied by some trigger activation. Due to our investigations, INSERT statements are here heavily affected by performance overhead. Furthermore, this environment cannot handle *Request*s because we are unable to get the query text in order to recognize what queries are complex. Manual work is still necessary.

Another proposal for staleness could be based on the number of physical blocks used for tables. This number can be computed for tables very cheaply. But on the one hand, the number of records must be derived from the block size – a block might contain 10 large or 1000 small records. On the other hand, the number of blocks is only a rough estimation as a DBMS will usually not release blocks in case of DELETE.

6. CONCLUSIONS

In this paper, we reported on performance problems in an existing application in the field of telecommunications. The application holds about 1000 tables with only few millions of records. Unfortunately, there are very complex queries running that join several tables and use aggregate functions. Those queries are often badly optimized and last minutes, but could be performed in milliseconds.

The reason for the bad query behavior is the cost-based query optimization. More precisely, performance problems arise when database-internal statistics are not up-to-date. In our application, it is not enough to compute these statistics in certain intervals because data is changing quickly and dramatically. It is thus difficult to find an adequate strategy for determining when to calculate database statistics, on what tables, on time.

Our solution to overcome these problems in an elegant way is to use the recent technology of aspect-orientation. Aspect-oriented languages such as AspectJ help us to provide some automatism. The main idea is to extract SQL statements and to determine the number of changed records. Using this information, we can decide when to compute database statistics and on what tables. Owing to AspectJ, this logic can be brought into existing applications by just writing an aspect without explicitly touching existing code. This approach resulted in performance benefits.

Indeed, the benefits and the urgent need to improve performance dissolve any fears about using a new technology. Another important point was that only JDBC code is intercepted. This is pretty comprehensible and does not endanger the applications.

The presented solution provides a reusable asset that can be applied to any JDBC-based application to improve performance; anyway the correct configuration of parameters has to be found. The solution has still some drawbacks. It works for one process only, i.e., the environment is not yet centralized; the statistical data is collected separately for each process. It could now happen that the modifications of one process on a table do not exceed the threshold, but the modifications of all processes would do. For instance, if four processes insert 4999 records each (just below the threshold), then no calculation of statistics would take place for any process although 19996 records have been inserted in total. And if all processes insert the missing record, then all of them would calculate the statistics although only one overall calculation would be enough. In case of our telecommunication scenario, this is no real problem since the most important processes are not working in parallel.

Having parallel processes, a central environment is required that maintains the changes for all processes. Unfortunately the environment must be a process of its own in order to collect numbers from all processes. It is still possible to use AspectJ for collecting changes as described above, and to pass the number of changed records to a CBO process by the presented `CBO_KEEPER` advices. But the overhead would be too high due to inter-process communication, especially if SQL statements are executed in a loop, notifying changes or requesting up-to-date statistics inside.

One idea to solve the loop problem is to collect the data internally (as it is done now), and to forward the collected numbers at the end of the loop only. But unfortunately AspectJ does not offer the possibility to trap loops. Hence, further elaborations are necessary in order to provide an adequate central environment.

7. References

[1] Aboulnaga, A., Haas, P., Kandil, M., Lightstone, S., Lohman, G., Markl, V., Popivanov, I., Raman, V.: Automated Statistics Collection in DB2 UDB. Proc. VLDB 2004

[2] Chaudhuri, S.: An Overview of Query Optimization in Relational Systems. Proceedings of 17th Symp. on Principles of Database Systems, Seattle, Washington, 1998

[3] Chaudhuri, S.; Motwani, R.; Narasayya, V.; Random, V.: Sampling for Histogram Construction: How much is enough? In Proc. of ACM SIGMOD, Seattle 1998

[4] Chaudhuri, S.; Shim, K.: An Overview of Cost-Based Optimization of Queries with Aggregates. IEEE DE Bulletin 1995 (Special Issue on Query Processing)

[5] Clarke, S.; Walter, R.: Composition Patterns: An Approach to Design Reusable Aspects. ICSE 2001

[6] Elrad, T.; Filman, R.; Bader, A. (eds.): Theme Section on Aspect-Oriented Programming. CACM 44(10), 2001

[7] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: Design Patterns – Elements of Reusable Object-Oriented Software. Addison-Wesley 1995

[8] Gassner, P.; Lohman, G.; Schiefer, K.: Query Optimization in the IBM DB2 Family. IEEE Data Engineering Bulletin, 1993

[9] Gupta, A.; Harinarayan, V.; Quass, D.: Aggregate Query Processing in Data Warehousing Environments. In Proc. of 21st Int. Conf. on VLDB, Zurich 1995

[10] Gibbons, P.; Matias, Y.; Poosala, V.: Fast Incremental Maintenance of Approximate Histograms. In Proc. of VLDB, Athens 1997

[11] Graefe, G.: Query Evaluation Techniques for Large Databases. In ACM Computing Surveys 25(2), 1993

[12] Haas, L.; Carey, M.; Livny, M.; Shukla, A.: Seeking the Truth About ad-hoc Join Costs. VLDB Journal 6(3), 1997

[13] Haas, L.; Naughton, J.; Seshadri, S.; Stokes, L.: Sampling-Based Estimation of the Number of Distinct Values of an Attribute. In Proc. of 21st VLDB, Zurich 1995

[14] Ioannidis, Y.; Ng, R.; Shim, K.; Sellis, T.: Parametric Query Optimization. Proc. of 19th Int. VLDB, Vancouver (Canada), 1992

[15] Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersten, M.; Palm, J.; Griswold, W.: An Overview of AspectJ. ECOOP 2001, Springer LNCS 2072

[16] Kienzle, J.; Guerraoui, R.: AOP: Does it Make Sense? The Case of Concurrency and Failures. ECOOP 2002, Springer LNCS 2374

[17] Laddad, R.: AspectJ in Action. Manning Publications Greenwich 2003

[18] Levy, A.; Mumick. L.; Sagiv, Y.: Query Optimization by Predicate Move-Around. In: Proc. of 20th VLDB, Santiago (Chile) 1994

[19] Ono, K.; Lohman, G.: Managing the Complexity of Join Enumeration in Query Optimization. In Proc. of VLDB, Brisbane 1990

[20] Poosala, V.; Ioannidis, Y.: Estimation without the Attribute Value Independence Assumption. In Proc. of VLDB, Athens 1997

[21] Rashid, A.; Chitchyan, R.: Persistence as an Aspect. In M. Aksit (ed.): 2nd Int. Conf. Aspect-Oriented Software Development Boston, ACM 2003

[22] Rashid, A.: Aspect-Oriented Database Systems. Springer Berlin Heidelberg 2004

[23] Seshadri, P. et al: Cost-Based Optimization for Magic Algebra and Implementation. In Proc. of ACM SIGMOD Montreal 1996

[24] Soares, S.; Laureano, E.; Borba, P.: Implementing Distribution and Persistence Aspects with AspectJ. OOPSLA 2002, ACM Press

[25] Tsois, A.; Karayannidis, N.; Sellis, T.; Theodoratos, D.: Cost-Based Optimization of Aggregation Star Queries on Hierarchically Clustered Data Warehouses. Proceedings of the 4th Intl. Workshop on Design and Management of Data Warehouses DMDW 2002, Toronto, Canada

[26] Viglas, S.; Naughton, J.: Rate-Based Query Optimization for Streaming Information Sources. ACM SIGMOD, Madison (Wisconsin) 2002

Using Aspects with Object-Oriented Frameworks

Michael Mortensen
Hewlett-Packard
3404 E. Harmony Road, MS 88
Fort Collins, CO 80528
and
Computer Science Department
Colorado State University
Fort Collins, CO 80523
mmo@fc.hp.com

Sudipto Ghosh
Computer Science Department
Colorado State University
Fort Collins, CO 80523
ghosh@cs.colostate.edu

ABSTRACT

We investigate potential uses of aspect-oriented programming in the context of object-oriented C++ frameworks used in the development of VLSI CAD applications. We use existing applications to explore the use of different kinds of aspects. We differentiate between framework-based aspects and application-specific aspects. Framework-based aspects modularize cross-cutting code based on how an application uses or extends an object-oriented framework. We propose the use of a library of framework-based aspects that can be developed for and leveraged across a family of framework-based applications. Application-specific aspects allow modularizing existing cross-cutting code in VLSI CAD applications. Preliminary results for each type of aspect are presented, along with challenges in identifying and using aspects.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.3 [**Software Engineering**]: Coding tools and techniques; D.2.13 [**Software Engineering**]: Reusable Software—*Domain engineering,Reusable libraries*

Keywords

Aspect-oriented programming, AspectC++, Object-oriented frameworks

1. INTRODUCTION

Object-oriented frameworks are important for the development of large-scale industrial applications because they provide a common library of functionality (by defining an application programming interface or API) and an object-oriented model of the domain (as a set of class hierarchies that can be extended) [30]. Object-oriented frameworks typically provide classes that application developers can use directly, or reuse through object-oriented mechanisms such as composition, inheritance, and polymorphism [27]. Aspect-oriented programming can complement object-oriented programming by modularizing cross-cutting concerns that do not fit into inheritance hierarchies or procedural programming. As Kiczales et al. [17] state, "the central idea of AOP is that while the hierarchical modularly mechanisms of object-oriented languages are extremely useful, they are inherently unable to modularize all concerns of interest in complex systems".

We are investigating potential uses of aspect-oriented programming in the context of object-oriented C++ frameworks. The framework being studied is an object-oriented C++ framework used in the development of VLSI CAD applications at Hewlett-Packard. Existing framework-based C++ applications are of interest for two reasons. First, using existing applications can help understand the types of cross-cutting concerns that exist in legacy C++ software, which is often a mix of object-oriented and procedural styles. Second, maintenance of existing application software is a key part of the lifecycle of industrial applications, with practitioners reporting that it consumes more time and more resources than any other part of the software lifecycle [31]. We are studying multiple framework-based applications to identify common aspects used by many frameworks.

This paper reports on the results of identifying aspects in two framework-based applications. We have identified two types of aspects: framework-based aspects and application-specific aspects. Framework-based aspects modularize cross-cutting concerns that are based on not only the application structure, but also how an application uses the framework classes and API. Application-specific aspects represent cross-cutting concerns that exist in these C++ applications but do not use classes or methods of the object-oriented framework. For the aspects identified, we report on the benefits (code reduction, increased modularity, improved defect detection) as well as some potential drawbacks of refactoring applications to use aspects. Aspects that use framework calls or framework-based pointcuts can be grouped together as an aspect library that is associated with the framework.

The remainder of the paper is structured as follows. Section 2 describes the process used for identifying aspects for use in object-oriented frameworks. Section 3 describes the applications that were studied and the framework they are based upon. Section 4 details an aspect created for de-

AOSD Industry Track 2006 Bonn, Germany

bugging a framework-based application. In Section 5, we describe an extra-functional Timing aspect. Section 6 discusses refactoring a large application to use a policy enforcement aspect. A discussion of aspects and frameworks is provided in Section 7. Related work is reviewed in Section 8, followed by our conclusions in Section 9.

2. IDENTIFICATION OF ASPECTS

Aspects are typically associated with duplicated (or very similar) code that is scattered throughout an application and tangled with its local context [18]. These duplicates are often a good starting point for identifying framework-based aspects and application-specific aspects.

Framework-based aspects may also be based on framework extension points. These extension points include framework classes that applications commonly extend to form class hierarchies, callbacks that applications can use to register code to be called when certain events occur, and functors (function objects) that can be overridden to provide application-specific behavior in framework code. Framework documentation typically identifies expected extension points for application developers. Coady and Kiczales [6] reported improved modularity and maintainability when aspects were used to implement extension points in operating systems code. Since extension points are used in many applications and may be tangled with application-specific code, they may be good candidates for searching for framework-based aspects.

The design documents and source code of an application can be inspected for classes that are infrastructure-related (e.g., logging, memory management, performance monitoring), rather than directly related to the main functionality (VLSI CAD) of the application [14]. Since infrastructure-related classes typically affect many parts of an application, they are candidates for aspectualizing. Infrastructure-related aspects could be application-specific, or they could be framework-based if they modularize cross-cutting concerns related to how an application uses the framework.

Once we identify a concern that might be better modularized as an aspect, simple tools such as code browsers and Unix utilities like `grep` can be used to quickly find other occurrences of similar or related code to see if it is really cross-cutting. For example, if a logging class is used, we can `grep` all files for its type name (e.g. "grep CadLogger *.cc"). This has limitations, since `grep` considers one line at a time. For example, we cannot use grep to find occurrences of two class names within 3 lines of one another without using a complex script. In addition, grep is purely text based, while many code editors (e.g. Eclipse[1], SlickEdit[2]) can display syntactic relationships for an entire project, such as class diagrams, function calls, and method calls. Visualizing more complex relationships can help identify methods that are called in many places or span hierarchies, and can also help determine if a concern is scattered enough to warrant creating an aspect.

Because of the large size and long weave-times associated with the framework-based applications we are studying, we have adopted a test-driven process that uses mock systems to prototype aspects before applying them to an application [25]. After prototyping an aspect with a small mock system, we refactor the application itself and run existing application tests to ensure the change has the desired effect.

The refactoring process can be summarized with the following steps:

1. Review framework documentation for extension points.
2. Inspect application code for duplicate, tangled code.
3. Identify any infrastructure-related code that may cross-cut many applications.
4. For each aspect, use a small mock system to develop and test a prototype.
5. Refactor the application to use the aspect.
6. Run application regression tests.
7. To enable reuse, create abstract and concrete aspect.
8. Add framework-based aspects to aspect library.

Application-specific aspects also use the above steps, except for the first one. Since all of our applications are in the VLSI CAD domain, even application-specific aspects could be considered for library inclusion, since they may be usable by other applications in this domain.

Like traditional refactoring, the aspects are intended to preserve the behavior or intent of the program constructs [11]. The aspects will become part of a framework-based aspect library that we are developing. In order for aspects to be usable by other applications, an abstract aspect is created for the library. An abstract aspect has a virtual, abstract pointcut, and may have virtual methods that are called by the advice. The application uses a concrete extension of this aspect.

We have identified both production aspects and developmental aspects [19]. Developmental aspects are used for testing, profiling, tracing, or debugging an application, but are not part of the production system. Production aspects are required for the application to function properly and are delivered as part of the system.

3. APPLICATIONS AND FRAMEWORK

We have studied two applications: PowerAnalyzer and ErcChecker. We describe each briefly, since all aspects in this paper were created refactoring these applications. Both use an object-oriented VLSI CAD framework developed at Hewlett-Packard. The framework provides a class hierarchy for the domain of electric circuits, as well as parsers for common file formats, collection classes and iterators, and utility classes for performing common operations. The framework uses inheritance, including mixin classes, to provide many concrete classes and allow developers to create their own extension classes.

The PowerAnalyzer tool is used for estimating power dissipation of electrical circuits. It consists of about 12,000 lines of C++ code. Power dissipation results from switching power (signals changing from 0 to 1 or vice-versa) and from static leakage power (current that leaks away even when signals are not changing). The PowerAnalyzer tool is composed of 3 related executables: PowerSrc, PowerCap, and PowerEst. PowerSrc checks the structure of the circuit and ensures that all needed connectivity and electrical data is available. PowerCap is used to calculate capacitance since

[1] www.eclipse.org
[2] http://www.slickedit.com/

power switching power is proportional to capacitance. PowerEst reads data generated by the PowerSrc and PowerCap programs and generates a power estimate for each signal of a block. All three programs use an application-specific library of common functions and classes (`libPower`) in addition to using the HP VLSI framework. Because they are not variants of the same program, they are not a product line, but instead constitute a product family [4]. The use of several small, focused programs is a common design style for Unix programs [13].

Electrical Rules Checking (ERC) systems automate a number of electrical circuit checks. Example checks include checking for proper transistor ratios between the pull-up and pull-down transistors of an inverter, checking for fan-out limits, or checking for drive strength problems [26]. An ERC checking tool typically performs many types of checks on a circuit, reporting violations to the circuit designer. In order to understand (and ultimately correct) the violation, the user (a circuit designer) needs access to contextual circuit data, which the tool displays and writes to a log file. The ErcChecker tool consists of approximately 80,000 lines C++ code.

4. A DEBUGGING ASPECT

The first aspect identified is a development aspect used in debugging the applicaton's use of the framework. Validation and defect removal have been identified as challenges of using frameworks, since it can be difficult to determine if a fault is due to a defect in the application, unexpected interactions between framework code and application code, or a defect in the framework [27].

The PowerAnalyzer tool, like many VLSI CAD applications, invokes framework methods to create a large graph (based on framework classes) that represents circuit elements (transistors, capacitors), circuit connectivity between these elements (called nets or nodes), and other properties. It then populates the graph with additional data and manipulates it. Framework-provided iterators are used to traverse and explore the electrical circuit (also called a design).

During development of the PowerAnalyzer, the framework indicated there was an unrecoverable error due to incorrect internal framework state. The framework code printed the name of the framework method being called and exited. Calling `exit()` in a C++ library makes debugging difficult since program state information is not captured before exiting. Although the framework error message included the name of the method that exited, the application had many places where iterators were used. Further complicating this was the fact that the framework provides many types of iterators (e.g. iterate over all nets in a component, iterate over all instances in a component, iterate over all ports of a net) that share a common base class, and the error message was generated from the base class.

Initially, we used the brute force approach of adding print statements to record the last call before `exit()` was called. This required modifying 18 locations in 4 files, finding the defect and fixing it, and removing the 18 modifications from the 4 files. By contrast, the same effect is achieved by a single aspect in AspectC++. The `CadTrace` aspect, shown below, utilizes the fact that all the framework iterators provide a `Reset()` method that is called to start the iteration of elements when tracing all framework iterator uses by the application:

```
#include <iostream>
using namespace std;
   aspect CadTrace {
   advice call("% %Iter::Reset(...)\")
   : before() {
      cerr << "about to call Iter::Reset for "
           << JoinPoint::signature() << endl;
   }
};
```

Because AspectC++ is a source-to-source translator, we created a separate `libPower` library that has the tracing enabled. Rather than having to unplug and plug in the aspect, we can link against the original code or the woven code to enable and disable tracing. Thus, if a similar bug is found later, we can simply switch to the traceable version of the library.

The `CadTrace` aspect depends only the framework: its pointcut and advice are not related to the application at all. Thus, it can be reused as is in any application that calls a framework Iterator. In addition, we can create an abstract tracing aspect that uses an abstract pointcut. Applications could then extend the abstract aspect and define the pointcut to match any framework methods.

5. AN EXTRA-FUNCTIONAL TIMING ASPECT

Aspects have been proposed for modularizing extra-functional concerns: functionality that, while important, is not part of the central task of an application [24]. Because run-times for VLSI CAD software can be long (hours or even days), a common extra-functional concern is to write time stamps to a log file before certain steps or to write the elapsed time after certain steps. The PowerAnalyzer does this by implementing a class, `Timer`, that has a method to reset the elapsed time and another method to return the elapsed time as a string suitable for writing to a log file. Before and after various steps for which the designers want timestamp information, instances of `Timer` are created and used.

Even though similar code to instantiate and use Timer objects exists at many locations, there was no common structure or naming convention for use by pointcuts in selecting where an aspect should be woven. To refactor the `PowerAnalyzer`, functions that used the `Timer` class had the `Timer` instance and calls removed, and were renamed from `FunctionName` to `tmrFunctionName` for pointcut selectability. Another challenge was that some of the application code was written as large, procedural functions. One of the programs, PowerEst, contained a 500 line function with 15 separate uses of the `Timer` module. These separate uses were either loop statements or conceptually separate code blocks. For this function, the Extract Method refactoring was first used ("Turn the fragment into a method whose name explains the purpose of the method")[11]. Creating a function with name that begins with `tmr` allowed capturing the join point in AspectC++, and using a meaningful name allowed the join point signature to replace a manually generated text description of what was being timed. For consistency, the same aspect was applied across all three programs (PowerSrc, PowerCap, and PowerEst).

The `Timer` functionality is encapsulated in an aspect, `TimeEvent`, in which we use around advice to create a `Timer`

instance, call `Reset`, and then we proceed with the original call. After the original function invocation, we call `CheckTime` to get the current time as a string, and write this through a `PowerEstimator`-specific logging function (`PrintI`). We also call `PowerMessage::WriteBuffers()` to ensure that all files and logs are written out immediately and are not buffered by the operating system. The AspectC++ implementation is shown below:

```
aspect TimeEvent {

   pointcut outer_tmr() =
      call("% tmr%(...)") && !within("% tmr%(...)");

   advice outer_tmr() : around() {
      //set up the timer
      Timer timer;
      timer.Reset();
      //proceed with the original call
      tjp->proceed();
      //write out the time used
      PrintI(911, "Time around %s: (%s)\n",
             JoinPoint::signature(),
             timer.CheckTime());
      PowerMessage::WriteBuffers();
   }
};
```

This aspect represents a reduction in code of the following 4 lines of code that were normally duplicated:

```
      Timer timer;
      timer.Reset();
      PrintI(911, "Time around %s: (%s)\n",
             JoinPoint::signature(),
             timer.CheckTime());
      PowerMessage::WriteBuffers();
```

In the PowerEst program, long functions would reuse the same `Timer` object. The aspect-oriented solution used one `Timer` per `Reset` call. Since 15 occurrences were factored out, the reduction was 15*3+2 = 47 lines of code. In place of manually using the `Timer` module, functions were created as pointcut targets. If creating a new function is thought of as adding 2 lines of code (one for the header or interface of the function, and one for the call), then 30 lines of code were added, for a total savings of 17 lines of code. In addition, one could argue that the resulting structure was also improved by using smaller, clearly named functions rather than a single monolithic function.

The PowerCap program, like PowerEst, contained a single monolithic function that used the `Timer` module seven times. It was also refactored into smaller functions. It had five locally defined `Timer` instances instead of only two, so the code savings when using the aspect was 7*3+5=26. Seven new functions were created, for an addition of 14 lines, resulting in an overall savings of 12 lines.

The PowerSrc program had only one function (main) that was timed, so it was refactored as:

```
int
tmrMain(int argc, char **argv, PowerMessage *pMsg);

int
main(int argc, char **argv)
{
  PowerMessage *pMsg = PowerMessage::GetMessage();

  return tmrMain(argc,argv, pMsg);
}
```

This change represents adding an extra layer of function call (3 lines) in PowerSrc (`main` calls `tmrMain`) and four lines of code are removed. Although the aspect only reduces one line of code in PowerSrc, using it ensures that all applications in the PowerAnalyzer product family use the `Timer` in a consistent manner.

For all three `PowerAnalyzer` programs the code savings was 30 lines. More importantly, all code for capturing and recording timer information is modularized into a single aspect. If the `Timer` were to be modified, or if a new timing module were substituted, the entire change could be made in the aspect and the underlying code would not be changed.

The style of name-based pointcuts results in tight coupling that could be inadvertently broken during maintenance due to name changes in functions [29]. Here, we have tight coupling between the `TimeEvent` aspect and the naming convention of methods. If a new function is added that should use the timer, it must begin with `tmr` or it will not have the `Timer` functionality woven in. In addition, if someone changes one of the names of the functions to no longer begin with `tmr`, time logging will no longer occur for that function. If a function that does not need timing information is created with a name that begins with `tmr`, that function will match the pointcut and have `TimeEvent` advice associated with it. Unfortunately, such problems would not be immediately detected since the regression tests for the PowerEstimator focus on functionality and not the non-functional, orthogonal concern of time logging [10].

One maintenance problem that we can avoid is a future direct use of the `Timer` module. Although AspectC++ does not provide a `declare error` construct like AspectJ's, C++ templates can be used to provide compile-time assertions by defining a template that has a boolean argument and only providing a definition for a true value. Code that instantiates the template with a false value will trigger a compile-time error [1].

Lohmann [21] suggested using a C++ template approach that can be combined with static join point information so that any calls to a function matching a pointcut will trigger an error. For the `TimeEvent` aspect, we can add the following advice so that any direct calls to the `Reset` method of the `Timer` class result in a compile-time error.

```
// create a template that has no definition
// when the second type value is 'false'
template <typename JP, bool>
  struct DirectCallOf__StartTimer;
template <typename JP>
  struct DirectCallOf__StartTimer<JP, true> {};

aspect TimeEvent {
   advice call("% Timer::Reset()") : before() {
    DirectCallOf__StartTimer< JoinPoint, false >();
   }
};
```

6. A POLICY ENFORCEMENT ASPECT

The `ErcChecker` implements 58 different electrical checks. Each one is a subclass of an abstract class ErcQuery, and must implement a set of common virtual methods, including `createQueries()` and `executeQuery()`. The `createQueries()` method is a static method similar to the Template Method design pattern [12], that is called on a particular circuit element, such as a transistor or electrical node. Since it is static, it is not associated with any single object; in this application, it is responsible for creating and evaluating objects of it's class type. The `ErcChecker` iterates over various circuit elements, repeatedly calling `createQueries()`, which creates query instances and evaluates them by calling`executeQuery()`.

For each query, `createQuery()` performs the same sequence of conceptual steps.

1. Use framework iterators and framework traversal methods to identify circuit data needed by the query.
2. For each part of the circuit where the necessary circuit data is found, create an instance of the specific electrical query class associated with the check.
3. Call the `executeQuery()` method to on the query object from the step above. This method will indicate if the electrical check found an electrical failure or warning, or if the circuit element passed the electrical check.
4. Add queries that result in a failure or warning to a container class, the `LevelManager`, which generates electrical reports and can be used by a graphical user interface.
5. Write the results of `executeQuery()` to a log file.
6. Delete queries that did not result in a failure or warning.

Several of these steps can vary according to the particular query class. In step 2 for example, some queries create and evaluate multiple query objects for a single circuit element while others create only one query object. Step 4 can be modified with a command-line argument to the program so that all query results are reported instead of only warnings and failures.

Although conceptually similar to the Template method, no code is shared between classes, and there are significant variations between each class because of differences in the types of data being gathered. Thus, each of the 58 query classes has a single large method for `createQuery()`. In addition, for Query classes where many query objects are created, the six steps are repeated, one at a time, for each individual object, and some local context information is shared between the steps and accessed.

We created a `QueryPolicy` aspect to refactor steps 4 through 6. The aspect is implemented as after advice for the call to `executeQuery()`. Steps 1-3 are very query-specific, and are not refactored to an aspect. The AspectC++ implementation for the `QueryPolicy` aspect is shown below:

```
aspect QueryPolicy {
   pointcut exec_query(ErcQuery *query) =
      execution("% %::executeQuery(...)")
      && that(query);

   advice exec_query(query) : after(ErcQuery *query)
   {
       if(gReportAll || query->errorGenerated()) {
          LevelManager::addQuery(query);
          gLog->log() << "Query error: "
                      << " type: "
                      << query->getName()
                      << " start element: "
                      << query->getStartName()
                      << query->getSummary()
                      << endmsg;
          query->logQueryDetails();
       }
       else {
          gLog->log() << "Query ok: "
                      << query->getName()
                      << endmsg;
          query->logQueryDetails();
          delete query;
       }
   }
};
```

6.1 Specific Refactorings

While refactoring the `ErcChecker` to use the `QueryPolicy` aspect, several types of changes were made to adapt to the aspectual behavior. Many involved simple removal of code being handled by the aspect. We describe those that were not simple deletions of code below, followed by a summary of how many classes were associated with each type of change.

6.1.1 Error-only queries

Before refactoring scattered code to an aspect, some classes had a much simpler `createQueries()` structure than others. For example, most queries could pass or fail, so the query had to call the `errorGenerated()` method after calling `executeQuery()`.

```
query->executeQuery();
if(gReportAll || query->errorGenerated()) {
       LevelManager::addQuery(query);
       gLog->log() << "Query error " ...
}
else {
       gLog->log() << "Query ok: " ...
       delete query;
}
```

For some query classes, if a particular circuit structure was found at all, it always indicated the presence of an electrical error. The query code was simpler, since it did need to check the result of `query->errorGenerated()` and can immediately add the query object to the `LevelManager` class, as shown by these 3 lines:

```
query->executeQuery();
LevelManager::addQuery(query);
gLog->log() << "Query error " ...
```

The simpler structure has the potential for a defect in the refactored system. The `query->errorGenerated()` method uses an object attribute that should be set by `executeQuery()`. If the `executeQuery()` method fails to set that attribute, the original code (which did not check it) would function properly, but the new, aspect-based code, which uses advice that always checks the attribute value would

not function properly. These classes were inspected to ensure that their respective `executeQuery()` methods set the required attribute. Thus, although refactoring to an aspect did not introduce a defect, it does illustrate a case where using an aspect's advice with many core concern classes could introduce a defect if methods in a class heirarchy do not consistently use state variables.

6.1.2 Multiple queries in createQueries()

For most queries, `createQueries()` creates and evaluates a single query object. For others, exactly two objects are created (one for the high voltage case, and one for low voltage). In addition, some query classes have the potential to find many possible violations from a single starting point and iterate over a variable number of cases. For example, a wire in a circuit may have many neighbor nets with which it shares an unacceptable amount of coupling capacitance. When refactoring a class to use the `QueryPolicy` aspect, we must ensure that all of the calls to `executeQuery()` are removed and handled instead by the aspect.

6.1.3 Query-specific logging

In addition to the standard query logging, some queries contain additional class-specific code that calls methods not inherited from the base class to record query-specific details used for validating and debugging the electrical checks being performed. Because this is class-specific code, it cannot be called from an aspect advice that uses only the base class interface.

This code was refactored into a new method, `logQueryDetails()`, and directly called by the policy aspect. The base class for the queries was modfied to provide an empty default implementation for queries that do not need this functionality. For classes that did need the functionality, the class had to define the `logQueryDetails()` method.

In the original code for classes that had query-specific logging, the code was structured like this:

```
query->executeQuery();
   if(gReportAll || query->errorGenerated()) {
       LevelManager::addQuery(query);
       gLog->log() << "Query error " ...
}
else {
       gLog->log() << "Query ok: " ...
       delete query;
}
/* QUERY-SPECIFIC CODE HERE,
      USING 'query' VARIABLE */
if(!(gReportAll||query->errorGenerated()))
    delete query;
```

The call to `logQueryDetails()` must be performed in the aspect, because if we leave the query-specific code in (represented by the comments in all caps), and then use the aspect, we end up with this structure:

```
query->executeQuery();
/* Aspect will be called here */
/* QUERY-SPECIFIC CODE HERE,
      USING 'query' VARIABLE */
```

At first glance, we see the reduction in code that the aspect provides, but we may not realize that we could have a problem. If the query does not indicate an error, then the aspect's advice will delete the query object. In that case, when control returns to the method, the query-specific logging code will make method calls with the `query` variable. Invoking a method call on a deleted object in C++ will result in erroneous program termination. By having the advice call `logQueryDetails()` before the object may be deleted, we will not have method calls on deleted objects.

Since adding an empty method to the base class is a one line change, we added the method directly to the C++ header file rather than using an AspectC++ introduction. Creating `logQueryDetails()` required copying and pasting the query-specific code into the method body and changing method invocations to intra-class calls. Thus, method calls on an object like this:

```
gLog->log() << " Coupling net: "
            << query->getCouplingNet()
            << "  capacitance: "
            << query->getCouplingCap() << endmsg;
```

are now part of the `logQueryDetails()` method that has an implicit query object:

```
void HighCouplingCap::logQueryDetails()
{
   gLog->log() << " Coupling net: "
               << getCouplingNet()
               << "  capacitance: "
               << getCouplingCap() << endmsg;
}
```

6.1.4 Missing log information

The `QueryPolicy` aspect is woven into all calls to `executeQuery()`, consistently applying the policy through advice. During refactoring, we found that one query failed to provide any information to the log file, but was otherwise correct. Changing to the aspect fixed this oversight, eliminating an existing subtle defect.

6.1.5 Unchanged queries

There were a number of queries that were not electrical rules queries, but instead were used to flag non-electrical failures, such as problems in the system itself (e.g., failure to open a log file and unexpected data structure value in the framework). Warning a circuit designer of these problems was accomplished by implementing them as sub-classes of the `ErcQuery` base class. Doing this allowed the system error queries to be written to error files and displayed in the graphical interface consistently with the electrical checks, leveraging existing functionality. The non-electrical error classes did not actually call the `executeQuery` method, but directly added objects to the system when system-related issues were found during a tool run.

Because these system error classes differed from the electrical query classes, the non-electrical classes were not modified to use the `QueryPolicy` aspect. Since the aspect uses the `executeQuery` method, which is never called for these, the aspect did not affect them at all. The presence of these classes reflects a sub-optimal design decision that fortunately was easy to work around. A better long-term change might be to extract a subclass that distinguishes between family of electrical queries and the various non-electrical system queries [11]. Legacy systems often have methods or classes that could be refactored for a cleaner design, as

demonstrated by the large number of documented refactorings. In general, sub-optimal design decisions could affect aspect design and pointcut selection.

6.2 Summary of changes

The types of changes made (as well as the simple case of only deleting code) are shown in Table 1. Because some of the 58 queries involved more than one change type, the numbers in this table add up to more than the number of query classes when the number of unchanged queries (13) from section 6.1.5 are taken into account.

Type of change	# Queries
Simple deletion of code	15
6.1.1 Error-only queries	8
6.1.2 Multiple queries in createQueries()	6
6.1.3 Query-specific logging	18
6.1.4 Missing log information	1

Table 1: Changes made for QueryPolicy aspect

6.3 System-wide challenges

In addition to specific query-related challenges encountered during refactoring, we also found three system-wide issues that affected correctness, testability, and the change process.

6.3.1 Accidental code duplication between aspects and the core concern

The `QueryPolicy` aspect deletes query objects that do not find electrical errors. In refactoring the code, the original calls to `delete` must be factored out, or both the advice code and query code will try to delete the same object. This would introduce a defect that results in program termination. The underlying issue here is that aspect-oriented refactoring is asymmetric, since the duplicate scattered (such as `delete`) must be manually removed, but the corresponding call to `delete` in the advice will be automatically woven in.

Debugging the root cause for multiple deletions of the same object requires understanding the interaction between the aspect and core concern. If traditional tools such as debuggers are used, code comprehension may be more difficult, since the developer will be examining the woven code, which contains core concern code, aspect code, and low-level constructs used by the AspectC++ implementation of an aspect.

6.3.2 Standardizing output and regression testing

The `QueryPolicy` aspect standardizes the log file messages that indicate pass or fail for each query. We consider this a benefit for long-term maintenance, but it does cause any regression tests that look at output logs to fail due to the (now standardized) output. For this application, only a few more than 100 regression tests look at the log files; most look at report files that were not changed. In general, this kind of one-time maintenance change can present a significant adoption cost for AOP on large projects with extensive test suites.

6.3.3 Compile times

For the `ErcChecker`, weaving and compiling the code takes 25 minutes. This makes developing aspects, checking for proper pointcut matches, and testing advice code a tedious process. In fact, the desire for a faster, iterative prototyping approach to be able to experiment with and validate aspects was part of the motivation for our use of mock systems and test-driven development [25]. By creating a small mock system that had the same basic query structure and naming conventions, we could experiment with different pointcut statements and advice.

6.4 Benefits

Using the `QueryPolicy` aspect represents a line reduction of about 8 lines per class. However, as discussed in Section 6.1, not all classes had the same number of lines removed. In addition, factoring out `logQueryDetails()` added one line to the base class. Each class that used it needed to define the method in its C++ header file, and then the method name, opening curly brace ({) and closing curly brace (}) added 3 more lines. Thus, this change added $1 + 4 * 18 = 73$ lines of simple boilerplate code. The total difference in lines for the changes was a reduction of 262 lines. Taking into account the additional lines added for `logQueryDetails()`, 335 lines were removed from 45 classes. The aspect code that replaces this code is only 21 lines long.

More important than the lines of code reduced, the aspect enforces a policy of what must *always* occur after `executeQuery()` is called. During development, there were cases where the policy was accidentally missed for a class, since it cannot be automatically statically enforced when scattered through the many `createQuery()` methods for the subclasses.

7. DISCUSSION

Using the two framework-based applications, three aspects were identified. The first, a developmental aspect, was used to trace calls into the underlying framework. We are also investigating the identification of production framework-based aspects.

The other two aspects are production aspects, but there were still some important differences. The `Timer` module in the `PowerAnalyzer` can malfunction (or fail to run) without invalidating the results of the tool. By contrast, the policy enforcement aspect for the ErcChecker is required for proper functioning of the tool,

Other developers who use the same CAD framework have been enthusiastic about using development aspects, but have expressed concern about migrating to a new paradigm and a dependency on an aspect weaver for production aspects. Evaluating the use of aspects on these large C++ applications is beneficial for understanding the possible costs, benefits, and risks of adopting a new technology. Migrating to new technologies is not without risk and needs to be done carefully. In fact, VLSI CAD software saw an early unsuccessful attempt at large-scale object-oriented programming in the mid-1980s which adversely impacted the schedule and performance of a market leader's products [20]. In addition to demonstrating benefits and costs of using aspects, studying AOP in the context of large-scale applications can help provide a smooth transition in industrial use.

8. RELATED WORK

8.1 AOP-based Refactoring

Advocates of aspect-oriented programming have begun enumerating AOP-based refactorings and evaluating the associated benefits and costs [18]. Coady and Kiczales [6] demonstrated the benefits of using aspects in operating system code by implementing four modules as aspects in an early version of FreeBSD and then observing the changes to those modules as they introduced the changes from two subsequent versions. Our work focuses on framework-based applications, and has the long-term goal of studying a set of applications for common aspects among them.

8.2 Product Lines

Batory et al. [2] propose replacing large framework hierarchies with a set of components that can be combined in a layered approach to build "product lines" – families of related applications. Mixin classes (multiple inheritance) and templates based on custom-designed flexible component classes are combined using a grammar-based approach that specifies compositions [3]. Their product line approach relies on completely replacing (or reimplementing) the framework with small components that can be composed in pre-defined, grammar-based ways across many layers. Our approach uses an existing object-oriented framework without modifying it. Instead, the framework-based applications are modified to use aspects.

Lohmann et al. [24] propose the use of domain analysis, which produces feature diagrams, which are then used to guide in the designing an 'architecture-neutral' system that will allow aspects to be woven in across multiple modules so that the non-functional properties can be configured across a set of product lines. Although we may use domain analysis to identify non-functional aspects, our approach deals with a set of applications that share a common framework. Some of the applications we are refactoring are implemented as product families; however, the overall set of applications are not a product line since they are not variations of the same functionality.

8.3 Long-term studies of aspect-oriented refactoring

Coady [6, 7, 8, 9] has studied how aspect-oriented programming can be used to refactor complex code for operating systems. Included in her work was a retroactive study where aspects were added to an early version of the operating system code, and then the changes were applied to the refactored version to see how well-suited aspects were for system evolution. We plan on carrying out a similar longitudinal study, but will do so for a set of framework-based applications that use an aspect library.

8.4 C++ Templates and Obliviousness

C++ provides a powerful template mechanism that can be used to generate classes for types at compile time. Alexandrescu [1] shows how policy classes in C++ can use templates to provide structures that, like aspects, have an interface but still allow users a means to extend the internal code structure. Lohmann, Gal, and Spinczyk [23] demonstrate that these mechanisms can be used to develop code with an aspect-oriented style, but without the obliviousness of aspects: everything must be explicitly instantiated through templates, which have to have the extension points designed in.

The main limitation of implementing aspects as templates and of policy classes is that the extension points of an aspect or the actual policy location of a policy class must be designed into the template [23]. By contrast, obliviousness in aspect-oriented programming can be used to extend and customize classes and frameworks in ways they were not explicitly designed for. The use of AspectC++ with templates has been explored by Lohmann, Blaschke, and Spinczyk [22].

Burke [5] points out that obliviousness allows needed orthogonal behavior to be layered in above or below the functiionality that the behavior crosscuts so that the layered functionality can easily be enabled or disabled as needed. In our approach, we only weave aspects into the applications and not the framework code. The layers used by our aspect library would be around the framework and inside the applications but not below the framework interface.

Recently, obliviousness has been criticized as inadequate when designing and implementing new systems, since it requires sequentialization of the process; that is, first the base code is developed, and then the aspects are written based on syntactical properties and structure of the base code. Sullivan et al. [28] propose instead to define an interface between the base code and aspects, so that both can be evolved without accidental changes in dependencies. When using aspects with existing frameworks, however, we believe that sequentialization is fine: the framework is already implemented and available, and applications are developed to utilize the existing framework.

9. CONCLUSIONS AND FUTURE WORK

This paper has presented initial work at identifying and using aspects in framework-based VLSI CAD software written in C++. Both development and production aspects have been identified and used. The initial results in terms of reduction of code and modularity (grouping together related code) show improvement with an aspect-oriented approach.

Future work will continue investigating identifying aspects and performing refactoring of framework-based applications. We have begun investigating clone detection tools such as CCFinder[3] [16] for finding concerns, and are also interested in aspect-mining tools. Unfortunately, many current aspect mining tools are based on Java rather than C++ [15].

In addition, we are interested in identifying development and production aspects that are based on the framework so that a library of framework-based aspects can be created and used with many framework-based applications. We believe this approach, which we refer to as aspectualizing a framework, allows aspects to enhance framework functionality. In addition, it will ease integration with and use of frameworks without modifying the framework itself.

10. ACKNOWLEDGMENTS

The authors wish to acknowledge the use of the open source tool AspectC++ (www.aspectc.org). They also acknowledge helpful feedback on the AspectC++ user group from Olaf Spinczyk and Daniel Lohmann.

11. REFERENCES

[1] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. AW C++ in Depth Series. Addison Wesley, January 2001.

[3]CCFinder v10.1.2

[2] D. Batory, R. Cardone, and Y. Smaragdakis. Object-oriented frameworks and product lines. In *1st Software Product-Lines Conference (SPLC1)*, 2000.

[3] D. Batory and B. J. Geraci. Composition validation and subjectivity in GenVoca generators. *IEEE Transactions on Software Engineering*, 23(2):67–82, Feb. 1997.

[4] D. Batory, R. E. Lopez-Herrejon, and J.-P. Martin. Generating product-lines of product-families. In *ASE '02: Proceedings of the 17 th IEEE International Conference on Automated Software Engineering (ASE'02)*, page 81, Washington, DC, USA, 2002. IEEE Computer Society.

[5] B. Burke. It's the aspects. *Java Developer's Journal*, 2003.

[6] Y. Coady and G. Kiczales. Back to the future: A retroactive study of aspect evolution in operating system code. In M. Akşit, editor, *Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003)*, pages 50–59. ACM Press, Mar. 2003.

[7] Y. Coady, G. Kiczales, M. Feeley, N. Hutchinson, and J. S. Ong. Structuring operating system aspects: using AOP to improve OS structure modularity. *Commun. ACM*, 44(10):79–82, Oct. 2001.

[8] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. *ACM SIGSOFT Software Engineering Notes*, 26(5):88–98, Sept. 2001. Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT symposium on Foundations of software engineering, Vienna, Austria.

[9] Y. Coady, G. Kiczales, J. S. Ong, A. Warfield, and M. Feeley. Brittle Systems will Break - Not Bend: Can AOP Help? . In *Proceedings of the 10th ACM SIGOPS European Workshop on Operating Systems*. ACM Press, September 2002.

[10] A. Colyer, A. Rashid, and G. Blair. On the separation of concerns in program families. *Technical report, Lancaster, COMP-001-2004*, 2004.

[11] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Aug. 1999.

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

[13] M. Gancarz. *The UNIX Philosophy*. Digital Press, December 1994.

[14] S. Ghosh, R. B. France, D. M. Simmonds, A. Bare, B. Kamalakar, R. P. Shankar, G. Tandon, P. Vile, and S. Yin. A middleware transparent approach to developing distributed applications. *Software Practice and Experience*, 35(12):1131–1154, October 2005.

[15] J. Hannemann and G. Kiczales. Overcoming the prevalent decomposition in legacy code. In P. Tarr and H. Ossher, editors, *Workshop on Advanced Separation of Concerns in Software Engineering (ICSE 2001)*, May 2001.

[16] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, 2002.

[17] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proc. ECOOP 2001, LNCS 2072*, pages 327–353, Berlin, June 2001. Springer-Verlag.

[18] R. Laddad. Aspect-oriented refactoring part 1: Overview and process. Technical report, TheServerSide.com, 2003.

[19] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning, 2003.

[20] J. Lakos. *Large-Scale C++ Software Design*, chapter 0, pages 1–18. Addison Wesley, July 1996.

[21] D. Lohmann. 2006. AspectC++ user's mail list: http://www.aspectc.org/pipermail/aspectc-user/2006-January/000872.html.

[22] D. Lohmann, G. Blaschke, and O. Spinczyk. Generic advice: On the combination of AOP with generative programming in AspectC++. In *Third International Conference on Generative Programming and Component Engineering (GPCE'04)*. ACM, 2004.

[23] D. Lohmann, A. Gal, and O. Spinczyk. Aspect-Oriented Programming with C++ and AspectC++ (Tutorial). In K. Lieberherr, editor, *Proc. 3rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2004)*. ACM Press, Mar. 2004.

[24] D. Lohmann, O. Spinczyk, and W. Schrder-Preikschat. On the configuration of non-functional properties in operating system product lines,. In D. H. Lorenz and Y. Coady, editors, *ACP4IS: Aspects, Components, and Patterns for Infrastructure Software*, Mar. 2005.

[25] M. Mortensen, S. Ghosh, and J. Bieman. A test driven approach for aspectualizing legacy systems. 2006. Submitted to Aspect-Oriented Software Development (edited by Sami Beydeda and Volker Gruhn).

[26] S. M. Rubin. *Computer Aids for VLSI Design*. Addison-Wesley VLSI Systems Series. Addison-Wesley, 1987.

[27] D. Schmidt and M. Fayad. Object-oriented application frameworks. *Communications of the ACM*, 10(40):32–38, 1997.

[28] K. Sulivan, W. G. Griswold, Y. Song, Y. Cai, M. Shonle, N. Tewan, and H. Rajan. On the criteria to be used in decomposing systems into aspects. In *In European Software Engineering Conference and International Symposium on the Foundations of Software Engineering*, 2005.

[29] T. Tourwé, J. Brichau, and K. Gybels. On the existence of the AOSD-evolution paradox. In L. Bergmans, J. Brichau, P. Tarr, and E. Ernst, editors, *SPLAT: Software engineering Properties of Languages for Aspect Technologies*, Mar. 2003.

[30] J. van Gurp and J. Bosch. Design, implementation and evolution of object oriented frameworks: concepts and guidelines. *Software Practice & Experience*, 10(31):277–300, October 2001.

[31] J. P. Zagal, R. S. Ahus, and M. N. Voehl. Maintenance-oriented design and development: A case study. *IEEE Software*, 19(4):100–106, 2002.

On using AOP for Application Performance Management

Kamal Govindraj, Srinivasa Narayanan, Binil Thomas, Prashant Nair, Subin P

Tavant Techonologies
3101 Jay Street, Suite 101, Santa Clara, CA 95054, USA
kamal.govindraj@tavant.com,
srinivas.narayanan@tavant.com,
binil.thomas@tavant.com, prashant.nair@tavant.com,
subin.p@tavant.com

Abstract. We discuss our experiences in using AOP in the field of application performance management that were gained during our development of an open-source J2EE performance management tool called InfraRED. Monitoring the performance of enterprise applications lends itself well as a crosscutting concern. Basic monitoring features can be implemented through call and execution pointcuts, and interesting features such as slicing statistics by caller context, identifying top-level APIs and computing call-trees can be elegantly solved through the use of cflows. Summarizing statistics by layer, gathering statistics based on parameter values and controlling the parts of the application that need to be monitored are problems that can be solved in a simpler and more declarative way through AOP compared to other techniques. AOP tools can help further by providing features in the areas of pointcut definitions based on monitor entry/exit events such as synchronized blocks and waits, offering better runtime weaving support and offering in-built support for reasoning about advices such as the time spent in executing them. Overall, we believe that bringing the full richness of an AOP language to application performance management allows us to implement a rich feature set, and offers a level of flexibility and customization that is harder to achieve with other approaches.

1 Introduction

Managing the performance of J2EE applications is a non-trivial task. There are usually several layers of software to deal with such as Servlets, JSPs, EJBs [1], JDBC [2], third-party persistence tools (such as Hibernate [3], TopLink [4], iBatis [5]), independent frameworks (such as Struts [6], Spring [7]) etc., and dealing with such a wide variety of software components makes it difficult to detect and analyze performance problems.

We discuss our approach to application performance management (APM) of J2EE applications that involves the use of AOP techniques to address these problems. Our experiences are based on our work in developing an APM tool called InfraRED [8] that employs popular AOP tools such as AspectJ ([9], [10]) and AspectWerkz [11] as the basis for solving several interesting challenges. InfraRED has been used in several projects at Fortune 500 customers and is available through an open-source license. It is fully non-intrusive and lightweight [12] (i.e. adds very little overhead) that makes it easy to use in production systems. Interesting features of the tool include capturing statistics at various level of aggregation (by vertical components as well as horizontal layers) and granularity (at method level or based on method parameter values), correlating statistics from various layers, capturing call trees, and dynamically changing the level of information gathered. We highlight and re-iterate the advantages of using a full-fledged AOP tool to implement these features throughout the paper.

1.1 Outline

In Section 2, we present an overview of the traditional approaches to application performance management. In Section 3, we motivate several requirements of a good APM system such as basic timing and usage information, slicing statistics by layers or by caller context or by API parameter values, gathering persistence tier statistics and tracing remote calls. We show how we implemented them, how AOP helps in solving these problems elegantly in a simpler way compared to other approaches. We also mention where AOP may fall short and motivate our design decisions when there were multiple viable choices. In Section 4, we discuss the AOP tools we used and motivate why we support multiple tools. In Section 5, we highlight the benefits that a full-fledged AOP tool can bring to APM that would be harder to implement through other techniques. We then discuss other related work, provide ideas for what more AOP tools can do in future for APM, provide some insights into acceptance of AOP technology in our user community, and conclude in Section 7.

2 State of the Art

Application developers have traditionally used a few different techniques to deal with performance management.

The use of profiler tools (such as OptimizeIt [13], JProbe [14]) is a common technique. While such tools are useful to debug problems at a detailed level, they add a lot of overhead that make them unsuitable for use in production environments. Since many performance problems are difficult to reproduce on non-production environments, these tools are thus limited in their utility. Also, such tools do not typically provide statistics aggregated or sliced in different ways and do not correlate information from different tiers that are both necessary for a holistic approach to APM.

Another traditional approach has been to explicitly insert calls to gather timing and usage information at specific points in the code. However, this approach is intrusive and relies on programmer discipline, and is also tedious and difficult to maintain. Clearly, performance monitoring is a crosscutting concern that is elegantly solved by the use of AOP techniques.

Recent commercial APM tools [15, 16, 17, 18, 19, 20] have addressed this issue by using bytecode instrumentation techniques (typically at class loading time) or using interceptors provided by application server frameworks to add their own proprietary performance monitoring code. InfraRED takes a slightly different approach. While the underlying mechanism may eventually involve some form of bytecode instrumentation, instead of writing a lot of custom code to decide what code instrument and how, or adding a lot of interceptors and wiring them together to add the performance management logic, InfraRED uses higher-level AOP tools that weave the performance management logic into the application. Apart from InfraRED and Glassbox Inspector [21, 22], we are unaware of any other products that use full-fledged AOP tools for APM. As explained throughout the paper and highlighted specifically in Section 5, by making full use of AOP, we achieve a level of simplicity, flexibility and customization that is not available in the tools following other approaches.

3 AOP in Application Performance Management

3.1 The Basics

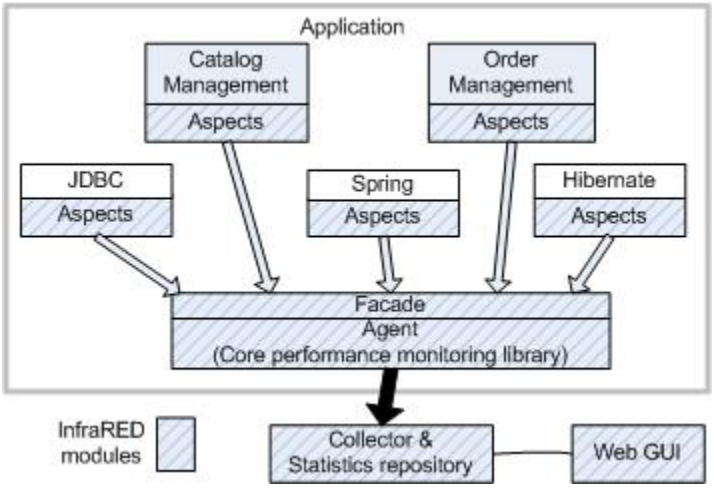


Figure 1: High-level Architecture

The agent consists of a core monitoring library which is a set of plain Java classes that calculates timing and usage statistics such as min, max, average, first, last execution times, creates call trees if necessary, and correlates these metrics with those collected from other parts of the system. The agent is bundled along with every application that is to be monitored. The agent periodically transfers the metrics it has collected to a central collector, which makes it available for a GUI to display to the user.

The agent exposes a façade [23] to which the application needs to make calls at significant points in its execution. These calls to the agent fit neatly into crosscutting concerns that can be weaved into the application code.

The main part of this mechanism is to define the aspects that capture the calls to the façade. We define an abstract base aspect as follows:

```
public abstract aspect
InfraREDBaseAspect {
    /**
     * The condition based on
which monitoring is
performed.
     */
     public abstract pointcut
operationToBeMonitored();

    /**
     * Gets the layer
(Session Bean/Entity
Bean/JDBC etc.) of
     * the operation
     */
    public abstract String
getLayerName();

    /**
     * Calls the agent façade
before and after the
execution of the
     * monitored operation
     */
    Object around() :
operationToBeMonitored () {
        final Class clazz =
thisJoinPointStaticPart.getSi
gnature()

.getDeclaringType();
        final String method =
thisJoinPointStaticPart.getSi
gnature()

.getName();
        final String layer =
getLayerName();

        long startTime =
Façade.start(clazz, method,
layer);
        try {
            return proceed();
        } finally {
Façade.end(startTime, clazz,
method, layer);
        }
    }
}
```

Concrete implementations of the abstract aspect provides pointcuts for specific operations that need to be monitored – for e.g., session bean calls, struts executions, entity bean calls etc. They also have to provide a "layer name" (such as "Session Bean", "Struts" etc.) that gets associated with the statistics that is collected. This allows us to provide interesting statistics such as the overall time spent in a layer that is useful for performance and scalability analysis. A concrete SessionBeanAspect is illustrated below:

```
public aspect
SessionBeanAspect extends
InfraREDBaseAspect {
    /**
     * Execution of a
SessionBean method
     */
    public pointcut
operationToBeMonitored() :
      execution(public *
javax.ejb.SessionBean+.*(..))
;

    /**
     * Gets the layer name
(Session Bean/Entity
Bean/JDBC etc.)
     * of the operation

     */
    public String
getLayerName() {
        return "Session
Bean";
    }
}
```

We also provide such pre-defined aspects for other common areas such as Struts executions, EntityBean executions etc., to make it easy for developers to integrate the tool into their system. Also, exposing the pointcut definitions to users allows them to easily customize the parts of the application that need to be monitored if they desire to do so.

The advice executed at these pointcuts includes gathering basic timing and execution statistics, layer-based statistics such as time spent in various layers (such as Web Layer, Struts Layer, Hibernate Layer etc.) that is explained in Sections 3.2 and 3.4. Detailed statistics such as call trees can also be captured optionally.

Overhead of Performance Monitoring Tools

A common concern among performance monitoring tools and AOP-based tools in particular is the performance overhead of such tools. Figure 2 illustrates basic performance characteristics of a sample typical Web application that involves servlets which invoke business logic through session beans. The application was instrumented with performance monitoring advice from InfraRED (advice capturing call trees was not included in these experiments). The number of methods indicates the number of executed methods that have been instrumented in a single request. See Appendix A for a description of the experimental setup.

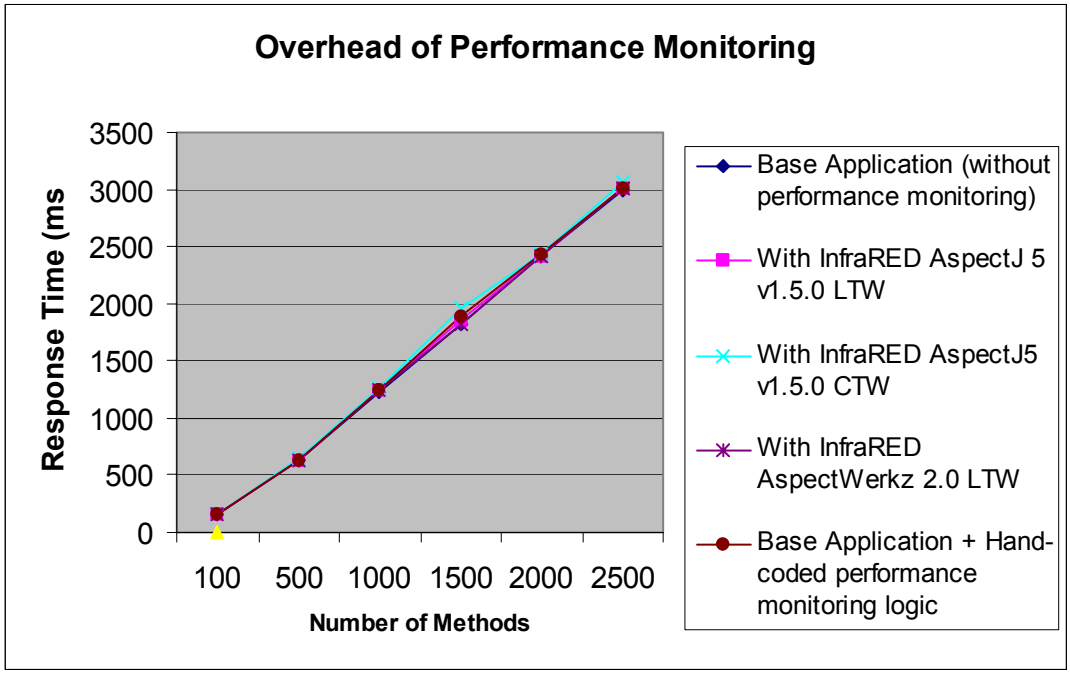


Figure 2: Overhead of Performance Monitoring

As we can see, the overhead of monitoring is minimal - less than 2% of the application's response time. And there is practically no difference when we compare the overhead of AOP-based performance monitoring to that of the hand-coded solution. From our experience with a wide variety of enterprise web applications, the overhead of performance monitoring in such applications using our tool has usually been about 1-5% of the application's response time, which is acceptable in most situations. From our experience, when we gather more detailed statistics such as call trees (see Section 3.4), the overhead ranges from to approximately 2-10%.

3.2 Slicing statistics by using cflow

The advice we described above allows us to view timing and call statistics for individual APIs as well as overall summaries for various typical layers. An important extension to this is the ability to view summary statistics for a layer further broken down by the type of the caller.

For example, consider a shopping cart application that uses Hibernate to access the relational store. Furthermore, suppose that CatalogMgmtService and

OrderMgmtService are services that handle the catalog management and order management portions of the application. A user might want to identify how much time the application spends in Hibernate just from the catalog management service to analyze any inefficiencies in how the CatalogMgmtService accesses the database. Using cflow pointcuts is a natural way to do this:

```
public aspect
HibernateInCatalogMgmt
extends InfraREDBaseAspect {

   /**
     * captures all hibernate
executions in the application
     */
  public pointcut
hibernate():
    execution (public *
org.hibernate.Session+.*(..))
      || (public *
org.hibernate.Query+.*(..)));

  public pointcut catalogMgmt
:
    execution (public *
com.mycompany.CatalogMgmt.*(.
.));

   /**
     * captures only the
hibernate executions used in
catalog management
     */
  public pointcut
operationToBeMonitored():
    hibernate() &&
cflowbelow(catalogMgmt);

  public String
getLayerName() {
       return "Hibernate In
Catalog Mgmt";
  }
}
```

A user integrating InfraRED with the shopping cart application can write such custom aspects and weave it into the application along with the other standard pre-defined aspects. As seen above, the logic is expressed in a simple and declarative way using a pointcut definition language. Achieving the same without AOP tools would involve writing a lot of cumbersome procedural code. The performance overhead of using cflow for this use case will also be minimal. We explain the performance considerations of using cflow more in Section 3.4.

3.3 Capturing statistics at a finer granularity

While recording timing and usage information at an API level is very common, some applications want to gather this information at a finer granularity based on the values of the parameters that were passed to the API. In our experience, we have encountered this need in cases such as monitoring performance of workflow tools and business rules engines. Many such tools have a generic API such as WorkflowEngine.execute(String workflowName) or RulesEngine.execute(String rulesPackageName) where there is a common API that takes the input of a workflow name or a package of rules and executes it. Applications typically want to see the timing and usage information broken down by the name of the workflow or the name of the business rules package. The core monitoring library in InfraRED supports gathering and displaying statistics for APIs based on the values of the parameters that were passed to them. An application just has to write a custom pointcut that describes the APIs in their system that need this feature, and invoke the core monitoring library from the advice for that pointcut. While this feature can also be built with other mechanisms, the AOP language provides a simpler and much more declarative way of using the parameters and types of APIs to express the necessary logic in the aspect definition.

3.4 More possibilities for using cflow

Layer time

InfraRED also supports capturing the total time spent in different layers of an application. A method is tagged with a layer name to which it belongs. Time spent in executing the top-level method of a layer is added to the overall time spent in that layer.

This is another feature that can be elegantly implemented with cflows. However, since the overhead of the cflow implementation was not acceptable in AspectJ version 1.2,

we originally implemented this feature through custom monitoring code. This implementation used a ThreadLocal map of layer names to "layer depth counts" that shows how many methods deep in that layer an execution stack is at any point. When the layer depth count reaches zero (i.e. when a top-level method of that layer finishes execution), the time spent in that layer is incremented.

Interestingly, AspectJ 1.2.1 introduced several cflow optimizations such as ThreadLocal counter based cflow implementations when there is no state with the pointcut, which is very similar to our custom implementation described above (but instead of a Map of layer names to counters, there is one ThreadLocal counter for each pointcut representing a layer). We measured these results on a sample application with 5 different layers (see Appendix A for a description of the experimental setup). With these optimizations, the performance of cflow based pointcut to compute layer timings is comparable to the hand-coded solution. Figure 3 shows the results of performance experiments of computing layer time through custom code and through use of cflows. Cflows in AspectJ 1.2 had an average overhead of 5% of the total application time over the cflow implementation in AspectJ 5 v1.5.0. The cflow implementation in AspectJ v1.5.0 itself turned out to be slightly better (about 1% of the total application time) than the custom implementation. We wish to note that this experiment is not a general statement about the efficiency of cflow implementations – it is merely the result of studying the use of the cflow technique to solve a specific problem in performance monitoring. This is also the reason why we compare the relative performance of these implementations against the response time of the base application with no monitoring.

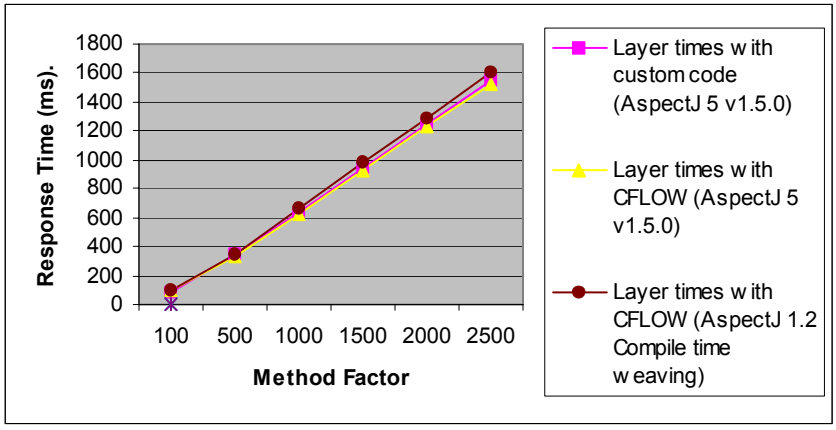


Figure 3: Overhead of cflow for computing layer times

Top-level APIs

We need to identify "top-level" APIs that are entry and exit points for a request into the system so that we can gather performance statistics for each request independently. Execution of the Servlet.do* methods or the execution of SessionBean methods are common top-level APIs. The following aspect can be used to gather the relevant statistics for top-level methods.

```
public aspect
RequestBracketingAspect {

  pointcut
potentialTopLevelMethods() :
execution(public *
    Servlet+.do*(...))
      || execution(public *
SessionBean+.*(..));

   pointcut topLevelMethods()
:
      potentialTopLevelMethod
&&
!cflowbelow(potentialTopLevel
Method);

   Object around() :
topLevelMethods() {
//Setup some data
//structure to start
//collecting data for
//this request
     Façade.startRequest();
     try {
```

```
      proceed();
    }
    finally {
      // Aggregate the data
collected for this request
      Façade.endRequest();
    }
  }
}
```

An alternate implementation for identifying top-level Servlet methods is to use a Servlet filter [24] to do the bracketing and use a thread local counter to ignore calls to a top-level method that happens in the context of another top-level method. We used Servlet filters to implement this feature due to a few logistical issues with instrumenting Struts libraries, but we are currently working on providing an alternate implementation using cflows.

Call trees

InfraRED provides support for capturing the call sequence of an execution. We implement this by writing custom code as part of the advice to create the tree structure and storing it in a ThreadLocal. However, this can also be achieved in a simpler manner by using a cflow construct as demonstrated in [22]. Use of cflow would reduce the complexity of the core module implementation.

To summarize, we have had to write a lot of custom code to implement some of the features described in this section that would have been simpler to implement using the cflow construct. When we started, cflow implementations were not optimized and our custom code was optimized for performance for the specific use cases that they represented. However, with the performance improvements in cflows, especially the availability of ThreadLocal counter based implementations for statesless cflow pointcuts, the difference in performance using cflows and custom code is negligible. However, for more advanced uses requiring cflow based pointcuts with state, we are yet to determine if the cflow implementation will have any significant overhead over handcrafted custom implementations.

3.5 Using call and execution pointcuts to track remote calls

A big issue with tracking remote calls across JVMs is the need to tag extra info with the remote call. JSR149 [25], Work Area Service for J2EE, is being developed by the JCP to address this need, but popular app servers are just starting to support it [26, 27]. Instead of attaching extra information with the remote call, which is difficult to implement without support from RMI/app servers, we instrument both the caller and callee – the caller is instrumented using a call pointcut and the callee is instrumented with an execution pointcut. We relate the two via just the method name and allow the user to drill down from the caller to the remote callee using the method name as a key, and thus be able to seamlessly analyze the performance of a system involving multiple remote calls and servers as a single unit. The pointcuts for the caller and callee are illustrated below:

```
public pointcut
remoteCaller() :
call(public *
java.rmi.Remote+.*(..))&&
target(Remote);

public pointcut
remoteCallee() :
execution(public *
javax.ejb.SessionBean+.*(..))
      || execution(public *
javax.ejb.MessageDrivenBean+.
*(..));
```

3.6 Issues with instrumenting third party libraries

Most J2EE applications involve significant amount of database access and applications are plagued by problems such as poorly performing queries, firing too many queries from the application for a request, and infrequent use of prepared statements.

P6Spy [28] is a popular library the uses the decorator pattern [23] to collect information about JDBC usage. We used a similar pattern using a combination of p6spy and custom written classes to monitor the JDBC usage and collect information such as time consuming queries, prepare-to-execute ratios, and correlate the JDBC calls to

relevant APIs in the application to make it easy to pin-point the source of problems within the application.

Since load time weaving (LTW) was not mature when we first release the tool, we tried to instrument the JDBC database driver and ship instrumented versions of the driver for our users, but this had a couple of issues. First, the JDBC driver jar file we used was signed and instrumenting it no longer makes the jar verifiable. We could get over this issue by just removing the manifest entry from the instrumented jar file. More importantly, we encountered a logistical problem where we found it very hard to enforce the usage of our instrumented JDBC driver because it interfered with the standard build and deployment procedures, policies and scripts of many applications. Since the p6spy driver was already available, we found that it provided an easy mechanism to implement our needs. However, this still required adding all the new decorator class implementations and it also requires setting up connection pools differently to use the p6spy driver at deployment time.

When LTW having become mature in AspectJ5, we wrote an aspect to instrument the getConnection() JDBC API to return a decorated connection. The decorated connection also decorates other relevant objects obtained from it (such as PreparedStatement) to gather JDBC statistics. This is an easy and effective way of obtaining JDBC statistics.

Some JDBC statistics that we gather requires us to maintain an association between a PreparedStatement and the SQL string used when creating it, and in some cases, also the association between the PreparedStatement and the values of the parameters used to execute that statement. Such information is not part of the regular JDBC APIs, and one possibility is to keep such associations separately in maps. But a more elegant object-oriented approach might be to keep the extra state along with the object where it belongs – such as adding extra fields to the PreparedStatement object to keep the original SQL statement that created it. We were able to do this using AOP ”mixins” in AspectWerkz 2.0 [11], but AspectJ5 1.5.0 doesn’t seem to fully support any analogous “mixin” construct.

3.7 A closer look at the advice: measuring overhead and exception handling

Measuring the overhead of monitoring performance in a production system is an important need. It would be nice if the performance management tool provided a means to report the overhead incurred by it. The aspect advice always invokes the core monitoring library (responsible for collecting and aggregating statistics) via a clearly defined façade interface. The overhead involved in performing this task can be measured by timing the calls to the façade. We have used a wrapper class based on the decorator pattern that intercepts calls to the Facade for capturing this metric. The wrapper also ensures that any exceptions that result due to errors in the core module will be suppressed and not be allowed to affect the actual working of the monitored application. The overhead of performance monitoring could have also been computed by adding an around advice to methods of the InfraRED façade.

We are currently working on using the statistics gathered about the overhead of the tool to dynamically tune the level of information gathered based possibly on specified tolerance limits (for e.g., a user might want to limit the monitoring overhead to at most 2% of the time spent by the application).

The decorator based approach is also used to avoid calls to the façade from within the core monitoring library to prevent infinite recursion. This would occur if the classes on which the core monitoring library depends on (for e.g., log4j [29]) were themselves aspected. Calls to log4j from with the application need to be monitored, but calls to the log4j from the InfraRED core monitoring module should be ignored. This can also be achieved by adding a !cflow (Façade.*(...)) to the pointcut definition.

```
public abstract aspect
InfraREDBaseAspect {
  /**
   * The condition based on
which monitoring is
performed.
   */
  public abstract pointcut
operationToBeMonitored();
```

```
  /**
   * Gets the layer (Session
Bean/Entity Bean/JDBC etc.)
of the operation
   */
  public abstract String
getLayerName();

  /**
   * Calls the agent façade
before and after the
execution of the
monitored operation
   */
  Object around() :
operationToBeMonitored()
                    &&
!cflow(execution (public *
Façade.*(...))) {
    final Class clazz =
thisJoinPointStaticPart.getSi
gnature()

.getDeclaringType();
    final String method =
thisJoinPointStaticPart.getSi
gnature()

.getName();
    final String layer =
getLayerName();

    long startTime =
Façade.start(clazz, method,
layer);
    try {
      return proceed();
    } finally {
      Façade.end(startTime,
clazz, method, layer);
    }
  }
}
```

However, our current implementation uses the decorator approach instead of using cflows for legacy reasons.

4 AOP Tools Used

When we started InfraRED, we used AspectJ 1.1 and its compile time weaving mechanism since that was the only mature AOP framework available. However, compile time weaving also introduced an additional step to the build process, often a troublesome one. For applications with a large code base, the instrumentation process was time consuming and memory intensive. The compile time weaving also made it harder to switch between the "aspected" version of the code and the original version.

With the AspectWerkz 2.0 release that provided good support for LTW, we extended InfraRED to work with AspectWerkz. LTW weaving helps us avoid most of these issues. Aspecting can be turned on or off easily by having parameterized startup scripts to launch the application with LTW turned on or off. The extra step in the build is not required, which enables shorter fix-build-test cycles. Since the pointcut / aspect definition is in an XML file, it can be modified easily which helps in easy experimentation.

We recently moved to AspectJ 5 (v1.5.0) that offers the flexibility of AspectJ and the LTW capabilities of AspectWerkz. However, unlike AspectWerkz, in AspectJ 5, we have lost some flexibility in defining new aspects through a XML file – for example, we are unable to define parameters and values when defining concrete aspects in the XML file. This feature provides the ability for a user of our tool to customize the performance monitoring of an application by adding csustom aspects with specific parameter values through a XML descriptor. This is a feature that is missing in AspectJ5 v1.5.0.

5 Better flexibility and customization through AOP

We now summarize and re-iterate how using the full power of a AOP tool allows for much higher level of customization, provide greater flexibility and makes it easy to add new features.

- Ability to capture statistics at the higher level of granularity: For e.g., it is easy to gather data about how much time was spent in the servlet layer, struts layer, persistence layer, JDBC layer etc. – AOP makes it easy to define these layers through a declarative pointcut definition language.
- Ability to capture statistics at a finer granularity (see Section 3.3).
- Flexibility in capturing more detailed statistics: With tools that do not expose the power of AOP,

users are stuck with the feature set that the tool provides. This makes gathering new statistics almost impossible. For example, some statistics that are very useful to have for APIs are frequency distribution, confidence intervals, first and last execution time, avg. time taken excluding the first execution (this is useful because some APIs do a lot of work on the first request such as reading a lot of data and caching it, and including the first execution time in the average frequently does not show a true picture of the time taken for an incremental request). InfraRED provides some of this statistics, but not all. Even if these features are not available in the core product, the use of an AOP tool gives users the power to add such custom advices on their own.

- Ability to slice statistics for a layer based on who called it (see Section 3.2).
- Ability to monitor only selected parts of the application beyond limiting them by just package or API names. The power of the pointcut definition language can be leveraged to do things such as monitor APIs only in certain execution contexts (using features such as cflow), monitor APIs based on parameters and their types etc.
- Ability to implement simple alternate mechanisms to drill through remote calls (see Section 3.5).
- Extensibility: An AOP language provides powerful declarative mechanism to add new features to a performance management system in a simpler and faster way than most other approaches.

It is also useful to point out the important areas in APM that an AOP tool does not directly address. AOP does not directly help in gathering data about hardware such as CPU and memory utilization, JVM runtime characteristics such as garbage collection statistics (frequency and elapsed time of GCs), thread info etc. – these are addressed very well in JDK 1.5 Monitoring and Management Platform [30]. It is also useful to point out that JDK 1.5 provides better infrastructure support for implementation of load time weaving and hot swapping.

6 How can AOP tools help further?

In this section, we provide thoughts for how AOP tools can evolve to address the specific needs of APM.

One common area for performance problems is resource contention and entry/exit from monitors. One typical source of this is the use of "synchronized" blocks to control access to common resources when accessed by concurrent users. A very useful statistic is to calculate the total time that an application spent just waiting for locks in synchronized blocks (probably broken down by the type of resource that it was waiting on), time spent executing synchronized blocks (this will give an idea for whether these monitors are too long-running which could reveal design issues in the application). In JDK1.5, JVMTI generates events on entry and exit to monitors that could be used to gather such statistics. However, the raw level of information from JVMTI may not be detailed enough for it to be put to full use by an APM tool – for e.g., we would like to have the wait time broken by class name or instance, or by caller context (e.g., how much waits happened with Hibernate or Struts code, or in my Shopping Cart module or in my Hibernate code within my Shopping Cart module etc.)

One possible way to implement these features would be to provide a pointcut definition language that allows for creating joinpoints based on entry/exit from synchronized blocks or other monitors – joinpoints before entering the block/monitor, just after entering the block/monitor and after exiting the block/monitor.

Another useful feature that an AOP tool can provide is to automatically report the amount of time spent in just executing the advice excluding any time spent in calling the original method that was weaved (for e.g., if we had an around advice on a method, we don't want to include the time spent in calling proceed(…)). When crosscutting concerns are implemented as advices, such statistics can be very useful to break down the performance of the system

into various layers and show the time spent in each crosscutting concern. AspectJ currently provides a total advice execution joinpoint and timing this joinpoint will also include the time spent in executing the original methods (i.e., including the proceed(…) call that executes the original method). Excluding such information and reporting these statistics would be a great value addition for APM tools.

Another area where AOP tools can improve is in offering better support for runtime weaving – i.e. weaving classes that have already been loaded. This would be very helpful for making the performance monitoring more adaptive and tuning the level of monitoring on the fly based on statistics gathered from the live system. While it is possible to incorporate such logic for adaptive monitoring within the Java code that is invoked from the advice, such a design limits the level of adpativeness that the system can offer. It makes it impossible to change the pointcut definitions based on real-time data – for example, we may want to start by just monitoring overall summaries for the JDBC layer and if we perceive a problem based on real data, we may want to break-down the statistics by the context in which the JDBC APIs are called which is easier to do by adding an extra pointcut definition based using cflow (see Section 3.2). There has been some work [31] in this area, but it needs to mature more and address the concerns mentioned therein.

Finally, we would like to see AOP help in easier performance management of systems consisting of multiple servers and JVMs, which is very common in enterprise applications. We would like to define pointcuts based on the patterns of execution across multiple VMs – for example, a session bean API called remotely may need to be monitored only if called from a specific caller (i.e., the ability to have "remote cflows"). We may also want to dynamically change the level of monitoring and the aspects that need to be applied at a remote callee based on data from the caller.

7 AOP Acceptance

Our customers always realized the attractiveness of AOP as a way to address crosscutting concerns and to avoid code duplication in their applications. Several of our customers have projects with large development teams (sometimes more than a hundred), and hence they are very eager to look for solutions that don't rely on the discipline of a large number of developers. While they agreed on the benefits of AOP conceptually, in the initial stages when we started the project about a year and a half ago, some of the users were wary of adopting a technology that was considered to be not mature – while they played with it in a development environment, they were reluctant to deploy it on production systems. They were also worried about deploying code that has been "altered", were concerned about performance, and worried about whether AOP will make bugs harder to reproduce and make debugging more difficult. Several of these concerns were addressed by training, doing performance studies of our tool [12] and educating people on AOP technology, how it works under the covers and their performance characteristics [32]. Also, in our opinion, the growing popularity in the industry of related technologies such as bytecode instrumentation (which is used in many performance management tools), dynamic code generation libraries (which is used in popular Java projects such as Hibernate), and some level of AOP support in popular application servers such as JBoss [33] have all led to an acceptance of AOP as a stable technology within our customer base.

8 Conclusion

Application performance management is a classic crosscutting concern that is well suited to the use of AOP. There are several requirements for an APM tool such as basic timing and usage information, separating statistics by layers, and tracing remote calls, and gathering statistics about the persistence tier, where AOP can be used as an effective implementation technique. The use of AOP also allows us to implement important features related such as statistics slicing (Section 3.2), gathering statistics for APIs based on their parameter values, and controlling what parts of the application to monitor in a simpler and much more declarative way than other techniques. There are also a few areas where we originally avoided the use of AOP mainly for performance concerns even when the use of AOP would have the solution much simpler and more elegant, and are currently re-

evaluating some of these decisions. Performance improvements in such areas, providing support for pointcut definitions based on entry/exit of monitors, runtime weaving, in-built support for reasoning about advices such as the time taken just by the advice execution, and cross VM AOP support are areas where the AOP community can help APM tools further. We believe that bringing the full richness of an AOP language to APM offers a level of simplicity, flexibility and customization that is harder to achieve with other approaches.

Appendix A

The test application is a typical servlet based web application. Servlets invoke some business code through a session bean and forward the results to a JSP page. By varying a few loop counter limits, we controlled the number of methods that were executed for a single request. The time taken to execute the requests varied from hundred milliseconds to a few seconds, which is typical of many web applications. When the application was instrumented with InfraRED, we configured the tool to gather performance statistics such as call and execution statistics and layer times. The overhead of capturing detailed statistics such as call trees was not measured in this experiment. The tests were run on a standard Windows 2000 desktop machine running a 2.8Mhz P4 processor with 1GB RAM. The application ran on Weblogic Server 8.1 on Sun JDK 1.4.

Acknowledgements

We thank all our colleagues at Tavant who participated in the development of InfraRED. We would also like to thank the AOSD conference reviewers for their useful comments on the paper.

References

[1] J2EE. http://java.sun.com/j2ee/

[2] JDBC Technology. http://java.sun.com/products/jdbc/

[3] JBoss Hibernate. http://hibernate.org/

[4] Oracle TOPLink. http://www.oracle.com/technology/products/ias/toplink/

[5] Apache iBatis. http://ibatis.apache.org/

[6] Apache Struts. http://struts.apache.org/

[7] Spring Framework. http://springframework.org/

[8] InfraRED. http://infrared.sf.net/

[9] AspectJ. http://eclipse.org/aspectj

[10] Laddad R. AspectJ in Action: Practical Aspect-Oriented Programming. Manning Publications Company. 2003.

[11] AspectWerkz. http://aspectwerkz.codehaus.org/

[12] InfraRED Performance Study. http://infrared.sourceforge.net/common/xls/InfraRED-PerformanceResults.xls

[13] Borland OptimizeIt Profiler. http://www.borland.com/us/products/optimizeit/

[14] Quest JProbe. http://www.quest.com/jprobe/

[15] Wily Introscope. http://www.wilytech.com/solutions/products/Introscope.html

[16] Quest PerformaSure. http://www.quest.com/performasure/

[17] Veritas i^3 for Java. http://www.veritas.com/Products/www?c=product&refId=315

[18] x.Link. http://www.abcseo.com/xlink/index.htm

[19] Trifork P4. http://trifork.com/products/P4/

[20] JInpired JXInsight. http://www.jinspired.com/products/jdbinsight/

[21] Glassbox Inspector. https://glassbox-inspector.dev.java.net/

[22] Bodkin R. Performance monitoring with AspectJ. In *IBM DeveloperWorks*. Sep 2005. http://www-128.ibm.com/developerworks/java/library/j-aopwork10/

[23] Gamma E. et al. Design Patterns, Elements of Reusable Object-Oriented Software. Addison Wesley Longman Inc. 1995

[24] Java Servlet Technology. http://java.sun.com/products/servlet/

[25] JSR-149: Work Area Service for J2EE. http://jcp.org/en/jsr/detail?id=149

[26] Websphere WorkArea Service. http://publib.boulder.ibm.com/infocenter/wasinfo/v5r1//index.jsp?topic=/com.ibm.wasee.doc/info/ee/workarea/concepts/cwa_overview.html

[27] Weblogic Context Propagation. http://e-docs.bea.com/wls/docs90/programming/context.html

[28] P6Spy. http://p6spy.com/

[29] Log4j. http://logging.apache.org/log4j/docs/

[30] Monitoring and Management for the JavaTM Platform, http://java.sun.com/j2se/1.5.0/docs/guide/management/index.html

[31] Vasseur A. Dynamic AOP and Runtime Weaving for Java - How does AspectWerkz Address It? In *Proc. AOSD*.2004.

[32] Hilsdale E., Hugunin J. Advice Weaving in AspectJ. In *Proc of AOSD* 2004.

[33] JBoss http://www.jboss.org

The Challenges of Writing Reusable and Portable Aspects in AspectJ: Lessons from *Contract4J*

Dean Wampler
Aspect Research Associates *and* New Aspects of Software
33 W. Ontario St., #29F
Chicago, IL 60610
dean@aspectprogramming.com

ABSTRACT

Contract4J is a developer tool written in AspectJ and Java that supports Design by Contract programming in these two languages. It is designed to be general purpose and to require minimal effort for adoption by users. For example, adoption requires little customization and prior experience with AspectJ. Writing Contract4J demonstrated several issues that exist when writing truly generic and reusable aspects using today's technologies. This paper discusses those experiences and comments on ways our understanding and tooling could improve to make it easier to write such aspects. In particular, I discuss the importance of migrating from syntax-based pointcut definitions to semantically-rich metaphors, similar to design patterns.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented Programming. Aspect-oriented Programming

General Terms

Design, Standardization, Languages, Theory.

Keywords

Aspect-oriented software development, object-oriented software development, design, AspectJ, Java, Contract4J, Design by Contract.

1. Introduction

Writing generic, reusable library software is difficult. This is no less true for aspect libraries, partly because of the relative immaturity of aspect design and programming techniques, but it also reflects the inherent nature of aspects themselves.

This paper discusses the lessons learned and challenges encountered while implementing *Contract4J* [3], a generic, reusable framework for *Design by Contract* (DbC) [6] in Java and AspectJ, which is written in AspectJ.

1.1 Design by Contract

All components have a "contract" for use, whether it is stated explicitly or not, DbC is an explicit formalism for describing the contract of a component and for automating contract enforcement during the test process. It is a tool for locating logic errors (as opposed to runtime errors like heap exhaustion). To remove the testing overhead, tests are turned off for production deployments.

A component's contract includes the input requirements that must be satisfied by clients who use the component, called the *preconditions,* and the constraints on the component's behavior (assuming the preconditions are satisfied), including *invariant conditions* and *postconditions* on the work done by the component (*e.g.,* method return values).

DbC also prescribes the rules for contract inheritance, based on the Liskov Substitution Principle (LSP [9]), which says that a class B is considered a subclass of class A if objects of B can be substituted for objects of A without program-breaking side effects. For DbC, this means that subclasses can only change the contract for their parents in particular ways. Invariants cannot be changed. Overridden preconditions can *relax* the constraints, because the client program will always meet a stricter subset of input constraints, namely the subset specified by the parent class. This is *contravariant* behavior, because while subclassing is a "narrowing" of sorts, the preconditions are "widened". In contrast, the postconditions can be narrowed, a *covariant* change, because the reduced subset of results will always satisfy the wider set of results expected by the client program, as stipulated by the parent class contract.

DbC was invented by Bertrand Meyer for the Eiffel language [6], which supports it natively. In addition to Contract4J, various toolkits have been invented that provide Java support through libraries or external tools. These include the XDoclet-based *Barter* package [2] and *jContract* [4].

Design by Contract complements Test Driven Development (TDD). Even if a developer relies exclusively on TDD, understanding the contractual nature of interfaces helps clarify design decisions.

2. Overview of Contract4J

2.1 Design Goals for Contract4J

Contract4J provides support for DbC in Java in an intuitive way and with minimal adoption effort. *Intuitive* means that users can specify component contracts using familiar Java features and they can do this efficiently and conveniently without obscuring the component's abstractions. Contract4J allows developers to embed contract information in the classes, aspects, and interfaces adjacent to the points where the contracts apply. This is a practical convenience for the developer and also keeps the contract portion of the component together with the component's methods and attributes, so clients have access to the full interface specification, of which the contract is an important part. The developer specifies the contract details in an intuitive format using familiar Java syntax, annotations or a JavaBeans-like convention that I call "ContractBeans".

Adoption includes straightforward build or load-time modifications and writing contracts as part of the usual development process. Hence, even developers without prior AspectJ experience can adopt Contract4J quickly.

2.2 How Contract4J Is Used

I illustrate using Contract4J with a simplistic bank account example. Figure 1 shows the basic interface.

```
interface BankAccount {
  float getBalance();

  float deposit(float amount);

  float withdraw(float amount);
  …
}
```

Figure 1: Simplified BankAccount Interface

There are methods for retrieving the current balance, depositing funds, and withdrawing funds. The balance-changing methods return the new balance. The interface is simple enough, but it leaves unanswered questions. What if the user tries to withdraw more money than the account currently has? What happens if the `amount` parameter in either the `deposit` or `withdraw` method is negative. Specifying answers to questions like these makes the full contract explicit. Consider Figure 2.

```
@Contract
interface BankAccount {
  @Post("$return >= 0");
  float getBalance();

  @Pre(amount >= 0")
  @Post("$this.balance ==
         $old($this.balance)+amount
         && $return == $this.balance")
  float deposit(float amount);

  @Pre("amount >= 0 &&
        $this.balance – amount >= 0")
  @Post("$this.balance ==
         $old($this.balance)–amount
         && $result == $this.balance")
  float withdraw(float amount);
  …
}
```

Figure 2: BankAccount with Contract Details

The `@Contract` annotation signals that this class has a contract specification defined. The `@Pre` annotation indicates a precondition test, such as a requirement on the `withdraw` method that the input amount must be greater than or equal to zero and it must be less than or equal to the balance, so that no overdrafts occur. Note that we can refer to the attribute `balance` that is implied by the JavaBean's accessor method `getBalance`, where the `$this` keyword tells Contract4J that `balance` refers to a field in the `BankAccount` instance being tested. The `@Post` annotation indicates a postcondition test, for example that the `deposit` or `withdraw` method must return the correct new balance and the new balance must be equal to the "old" balance (captured with the `$old(…)` expression) plus or minus the amount, respectively. Not shown is an example `@Invar` annotation for invariant conditions, which can be applied to fields, methods, or classes. The field and class invariants are tested before and after every non-private method, except for field accessor methods and constructors, where the invariants are evaluated after execution (to permit lazy evaluation, *etc.*). Method invariants are tested before and after the method executes.

The original interface plus annotations specifies the behavior more fully by explicitly stating the expected behavior.

This example shows the syntax supported by the latest version of Contract4J. In this version, Contract4J uses Jakarta Jexl (Java Expression Language) [5], a runtime expression evaluator, to evaluate the test strings in the annotations. This happens in the context of aspects that advice locations where the annotations are used. Typically, before advice is used for preconditions, after advice is used for postconditions, and around advice is used for invariants[1]. If a test fails, an error is reported and program execution halts.

A second experimental syntax uses a JavaBeans-style naming convention, which I call "ContractBeans". Using this format, the `BankAccount` interface is shown in Figure 3.

```
abstract class BankAccount {
  abstract public float getBalance();

  boolean postGetBalance(float result) {
    return result >= 0;
  }

  abstract public
  float deposit(float amount);

  public boolean preDeposit(float amount) {
    return amount >= 0;
  }
  public boolean postDeposit(float result,
                             float amount){
    return result >= 0 &&
           result == getBalance();
  }

  abstract public
  float withdraw(float amount);

  public boolean preWithdraw(
      float amount) {
    return amount >= 0 &&
           getBalance() – amount >= 0;
  }
  public boolean postWithdraw(
                           float result,
                           float amount) {
    return result >= 0 &&
           result == getBalance();
  }
  …
}
```

Figure 3: "ContractBeans" Format

This version does not support the "old" construct for remembering a previous data value, so the contract tests shown are slightly less precise than in the previous example (*e.g.,* `result >= 0`, instead of the more accurate `result = $old(result) + amount`). Also, I have switched to declaring an abstract class so that the tests, which are now defined as instance methods, can be defined "inline". An alternative would be to use an aspect and intertype declarations to supply

[1] Sometimes different types of advice are used in certain cases, for technical implementation reasons, as discussed later.

default implementations of the test methods for the original interface.

Following a JavaBeans-like convention, the postcondition test for the `withdraw` method is named `postWithdraw`. (Compare with the JavaBeans convention for defining a `getBalance` method for a `balance` instance field.) This method has the same argument list as `withdraw`, except for a special argument at the beginning of the list that holds the return value from `withdraw`. The `preWithdraw` method is similar, except that its argument list is identical to the `withdraw` argument list. All the test methods return `boolean`, indicating pass or fail.

This version of Contract4J uses runtime reflection to discover and invoke the tests. It was implemented as a way of eliminating issues with the original version of the annotation-based approach. However, the extensive reflection imposes significant runtime overhead and writing the tests is a more verbose process with a less "obvious" association between the tests and the elements they are testing.

The original version of the annotation-based approach did not use a runtime expression interpreter. Instead, it used a precompilation step to generate very specific aspects for each test with the test string converted to Java code. A custom plug-in for Sun's Annotation Processing Tool (APT) was used to find the annotations in the source code and to generate AspectJ aspects for each one, before compilation. This implementation is the simplest of the three versions, with excellent performance, but the precompilation step is a barrier to adoption. The expression interpreter version eliminates this issue, but the implementation is more complex internally, in part because it uses reflection, as I will discuss in detail below. Hence, it has a higher runtime overhead than the APT version. However, because Contract4J is a development/test tool, the performance is acceptable.

The following summary compares the strengths and weaknesses of the implementations. More details are provided in the subsequent sections. For completeness, I also include pros and cons for two alternative ways of doing DbC, simple Java `assert` statements and *ad hoc* aspects, such as those used as examples in some of the AspectJ literature.

"ContractBeans" Version

Pros

- Could be used with Java 1.4 and earlier code, since it doesn't use annotations.
- Tests are written as regular Java methods, which can be reused outside of Contract4J.
- Because tests are normal methods, they are checked by the compiler and IDE for typos and other bugs.
- If the tests are declared public, they are a visible part of the interface for clients and subclasses to see.
- The JavaBeans-like convention follows a metaphor familiar to developers.

Cons

- Significant runtime overhead for extensive reflection calls.
- Tests are somewhat verbose, because of the method "boilerplate", compared to annotations.
- If the tests are not declared public, they are not a visible part of the interface for clients.
- The JavaBeans-like convention has a few idiosyncrasies that can result in the tests being ignored. There is no mechanism to warn the user when this happens.

Annotations, Version 1 (APT Preprocessor)

Pros

- Most intuitive and succinct way of specifying contracts.
- Most flexible use of annotations, including tests on method parameters.
- Test inheritance follows correct behavior for Design by Contract, not the rules for Java 5 annotation inheritance, *i.e.,* method tests are inherited, even though method annotations are not.
- Contracts are properly part of the public interface for clients, including Javadocs.
- Fastest performance.
- Although tests are defined as strings, because they are converted to compiled AspectJ code, test syntax errors are caught by the compiler.

Cons

- Preprocessor step requires nontrivial build changes, which may not work well with IDEs and other tools.
- Since tests are defined in annotations, they are not easily reused in other ways.
- Although test syntax errors are caught by the compiler, the error messages point to the generated aspects, not the original annotations. The user must manually "map" the errors back to the original annotations.

Annotations, Version 2 (Jexl Interpreter)

Pros

- Most intuitive and succinct way of specifying contracts.
- Contracts are properly part of the public interface for clients, including Javadocs.
- Easiest adoption process; only minor build modifications required.
- Good performance.

Cons

- Can't use annotations on method parameters (not supported by AspectJ; but there are workarounds).
- Because test annotations are evaluated at runtime, tests defined on methods are not inherited automatically, following the inheritance and runtime-visibility rules for Java annotations. Subclass method overrides *must* include the same annotations *manually*. Class invariant annotations are inherited, although putting them on subclasses, for consistency, is harmless.
- Idiosyncrasies of Jexl expression interpreter complicate test writing slightly. (Read the examples and Contract4J unit tests!)
- A minor build change still required, *i.e.,* compiling or at least weaving with AspectJ.
- Since tests are defined in annotations, they are not easily reused in other ways.
- Since tests are defined as strings, they are not checked by the compiler or IDE for obvious test bugs. Buggy tests show up at runtime as Jexl expression failures with unintuitive error messages.

***Ad Hoc* Aspects** (Aspects hand written to test specific cases)

Pros

- Straightforward with no need to adopt a third-party toolkit, like Contract4J, if you are already using AspectJ.
- Complete flexibility to define arbitrarily complex tests and to define them in separate files, if desired.
- Tests are checked by the compiler and IDE.
- Optimal performance.

Cons

- Extensive, repetitive boilerplate code required that is handled automatically by Contract4J.
- Harder to present the complete interface specification to clients.
- Can clutter code being advised. Putting test aspects in separate files is possible, but that approach decouples the test "specifications" from the code, making the full interface specification obscure.
- Requires active use of and expertise in AspectJ.

Java Asserts

Pros

- Simplest way of specifying contracts.
- No AspectJ or other 3rd-party toolkits required.

Cons

- Slightly more invasive in the code.
- No coherent view of the contract.
- Not part of the client-visible interface.
- Not visible to other tools.

All three implementations of Contract4J share a common limitation; they only partially enforce the rules for contract inheritance discussed previously. Both the ContractBeans and APT annotation versions will invoke parent-class tests, unless overridden in subclasses. Because Java annotations on methods are not inherited, the Jexl annotation version cannot apply the tests for a parent-class method to a subclass override *unless* the override has the same annotations[2]. (However, the override can omit the test string; Contract4J will locate the parent's test string.) In contrast the Jexl/Annotation form does better at ensuring that invariant tests are not changed by subclasses. None of the three versions ensures that subclass preconditions are contravariant and postconditions are covariant.

Overall, the Jexl annotation version offers the best compromise of features and ease of use.

AspectJ is used in all three versions, but the aspects, while conceptually similar, are very different in the two versions. In the annotation-based version, since a precompilation step is used, all the aspects involved are generated during that step. They have very specific pointcuts, with no wildcards, that pick out just the join points for which a particular test is defined. These aspects are simple, although there can be a lot of them in a system with many DbC tests defined. However, because they are so specific and because they use no reflection, they have low runtime overhead[3].

For example, here is a simplified version of the generated precondition test aspect for the `withdraw` method.

```
public aspect BankAccount_pre_withdraw {
    before (BankAccount ba, float amount):
        execution (float BankAccount(float))
          && this(ba) && args(amount) {
        if (amount >= 0 &&
          ba.getBalance() - amount >= 0) {
        handleFailure("…");
    }
}
```

Figure 4: Example Aspect 1

In contrast, because the ContractBeans version eliminates the precompilation step, all logic has to be embedded in the runtime engine. This means that more complicated and comprehensive aspects are required to advise *all* possible join points for which a test might exist. The corresponding advice then uses runtime reflection to discover the tests, if any, and to invoke those that are found. Even if no tests are present for a particular join point, the overhead still exists.

All the pointcut definitions (PCDs) in this version are scoped by a marker interface (no annotations are used to permit use with pre-Java 5 source code). No class will be advised unless it implements this interface. Rather than explicitly adding this interface to all class and interface declarations, it is usually easier to write a custom aspect that uses intertype declaration (ITD) to add this interface into the classes of interest, as shown in the example in Figure 5.

```
public aspect EnableContracts {
  declare parents: com.foo.bar..*
    implements ContractMarker;
}
```

Figure 5: Aspect ITD of a Marker Interface

I discuss the two Annotation implementations and the ContractBeans implementation because each exposes different challenges for writing generic, reusable aspects that involve non-trivial interactions with the advised classes. However, for practical use, the ContractBeans implementation is considered experimental and is not recommended for normal use. I will explore the details and issues of these implementations in greater detail below.

3. Challenges in Aspect-Oriented Software Development with AspectJ

Most example AspectJ aspects you see are either very specific, using pointcuts that reference particular classes, methods, and fields (*e.g.,* Figure 4), or they are very general, using pointcuts that reference package hierarchies and/or class and method names with wildcards. Examples of the former tend to be tightly coupled to the advised classes, such as policy enforcement aspects to ensure proper usage of libraries, *etc*. The latter aspects usually implement *orthogonal* concerns, which means they have loose or

[2] This is a possible future extension. It could be implemented using reflection, but with significant overhead.

[3] Because DbC is primarily a development tool and the tests are (usually) removed from production builds, performance is not a serious concern anyway, as long as it is "reasonable".

no coupling to the classes they advise. Examples include tracing and authentication wrappers.

The main issue this paper addresses is the difficulty of writing closely-coupled aspects in a generic and portable way, *e.g.,* without embedding target-specific details in the pointcuts.

Let us delve into the issues in more detail, starting with a discussion of some general issues with Aspect-Oriented Software Development itself, which is still a young discipline, where many details of good design and coding practice need further development.

3.1 Conceptual Issues with Aspects

One of the interesting differences between aspects and objects is the scope of a "component" in each approach. Well-designed objects have a limited scope with minimal *coupling* to objects outside of their "namespace" or package. They also have high *cohesion,* a well defined and focused purpose and conformance to appropriate global and local conventions that contribute to system-wide "coherence" and consistency.

Well defined aspects should also have these properties internally, but because they are explicitly designed to support cross-cutting behavior, their coupling to other components is more complicated. Aspects that cause nontrivial changes of state and behavior to these components require new thinking about the nature of "interfaces" between the aspects and the components they advise.

When attempting to design generic, reusable aspects, this issue leads to a conundrum. For an aspect to offer fine-grained and powerful functionality, it needs some detailed information about the components it will advise. However, these details increase coupling to those components and reduce general applicability and reusability. Typical pointcut definitions written today rely on naming conventions and other syntax constructs used in the advised components, rather than relying on higher-level abstractions.

This leads to what I call a *concern semantics mismatch.* Component field, method, and class names reflect the primary concern of the component, the *dominant decomposition* [7], and they are likely to change as the problem domain understanding and/or the scope of the solution evolve. Pointcuts are part of a different domain, that of the cross-cutting concern, yet they are relying on the unrelated names and conventions in the components they advise, whose evolution will be "unexpected", from the concern's perspective, leading to fragile interdependencies.

The long-term solution is the development of higher-level design abstractions. The aspect-component relationship should be more of a "peer" relationship like the one that exists between objects today, rather than the approach commonly used where the aspect is "doing something" to another component. The noun "advice" and the concept of *obliviousness* reflect this bias, unfortunately.

Much of the research on aspect-oriented design (AOD) occurring now is moving away from this emphasis on oblivious insertion of advice and moving towards interface-based design approaches, *e.g.,* Aspect-Aware Interfaces [7] and Crosscutting Programming Interfaces (XPI) [8]. A compromise design strategy is emerging, where components will need to be "aspect aware", in the sense that they will need to expose state and behavior of potential interest to "clients", aspects as well as objects, without actually assuming particular details about those clients. The art of aspect-aware interface design will be to expose abstractions that are easily adapted by concerns that are different from the component's primary domain. I expect that most aspects will implement the *Bridge* pattern [10], connecting exposed interfaces with concern libraries. In fact, most aspects today follow this model, just in a more *ad hoc* fashion and with coupling to the fragile details of the advised classes, rather than coupling to more abstract and therefore stable interfaces. In other words, AOD is now expanding the established principles of object interfaces to support the new and unique needs of aspects.

3.1.1 Contract4J as a "Design Pattern"

You can view the annotation and the ContractBeans forms of Contract4J as *syntactically* different, yet *semantically* equivalent forms of an *ad hoc* "protocol", essentially a design pattern, which is used by a class to provide a design-pattern protocol for specifying the module's contract in a way that makes minimal assumptions about interested "clients" [6]. While invented for Contract4J, this protocol could be supported by a variety of other compile-time and run-time tools, including documentation tools and testing tools that generate unit tests from the annotations. The protocol is a mini domain-specific language (DSL) for DbC and it is conceptually consistent with the work on interface-based design in aspect systems [7-8]. In fact, a fruitful exercise would be to recast Contract4J in XPI formalism, for example.

3.2 Practical Challenges with AspectJ

Returning to AspectJ, its pointcut language is very powerful, but until recently, it has relied exclusively on concrete naming conventions, leading to the *concern semantics mismatch.* However, Java 5 annotations are a useful first step towards defining "interfaces" that support other concerns. Well-chosen annotations provide meaningful metadata about the element that tends to be more stable than naming idiosyncrasies of the element itself. Also, useful metadata will express information of interest to other concerns, implemented as aspects, in a more decoupled fashion. AspectJ 5 supports PCDs that match on annotations. Using annotations in Contract4J makes it unnecessary for it to know specific details about the classes it advises.

Put another way, most reusable aspects that have been documented to date are really reusable aspect *patterns.* They require customization of the PCDs to match on specific naming conventions for the project in question. The advices may also require modification. Truly generic PCDs that consist of almost all wildcards are often too broad, needlessly affecting far more join points than are really required.

However, having just made the argument that we need higher-level abstractions, it must be said that the lower-level join-point matching constructs currently available are still essential. Contract4J would not be possible without them. While annotations are used as "markers" for tests and for defining the test expressions, all the PCDs used in Contract4j still do matching on method and constructor calls or executions and field "gets" and "sets". This is in part an idiosyncrasy of Contract4J, since it supports detailed assertions about the component logic and those assertions have to be evaluated at very specific join points. Many other aspect-based tools and components will continue to require the lower-level constructs.

Let us consider the specific issues encountered in the three versions of Contract4J.

3.2.1 Contract4J Using Annotations, Version 1

Ironically, the original annotation-based version of Contract4J did not use any annotation-based PCDs. The precompilation step used a plug-in for Sun's Annotation Processing Tool (APT) to extract the annotation information and generate AspectJ code with PCDs that match on the specific classes, fields and methods with tests. The actual annotations are ignored in the PCDs, as they are no longer needed.

Figure 4 showed a simplified version of a typical aspect generated by this implementation. It uses the lower-level join point matching constructs, based on specific and explicit element names, because one aspect is generated for every annotation found (potentially creating a lot of aspects). This implementation proved to be the most straightforward to develop, because it did not require the more sophisticated PCDs needed in the subsequent Jexl aannotation version of Contract4J nor the more sophisticated introspection required in both the Jexl version and the ContractBeans version.

In fact, using a preprocessor tool (APT) avoided all the problems of the subsequent two versions of Contract4J, because using APT, a tool specific to the "annotation domain", if you will, handled all the dirty work of finding annotations and their context information.

3.2.2 Contract4J Using Annotations, Version 2

This is the most recent version of Contract4J and it is the one that will be maintained going forward. It uses annotation-based pointcuts to find the contracts and then uses the Jakarta Jexl expression interpreter [5] within advice to evaluate the test expressions at runtime.

Of the three implementations, this one has the most sophisticated aspects, combining nontrivial PCDs and construction of test context data that is passed to Jexl. The latter process uses Java's and AspectJ's reflection libraries to fill in information that can't be "bound" by the PCDs. In fact, the bulk of the code exists to support collecting context data and passing it to Jexl. The static typing of Java and the lack of "native" support for scripting (dynamic generation and evaluation of expressions) greatly complicated the implementation.

Consider two example aspects from this version. The first aspect implements method precondition tests and the second implements field invariants for field reads and writes.

3.2.2.1 Aspect for Method Precondition Tests

The PCD for this aspect is shown in Figure 6[4]

```
pointcut preMethod (                    // 1
  Pre pre, ContractMarker obj) :        // 2
    if (isPreEnabled()) &&              // 3
    !within_c4j() &&                    // 4
    execution (@Pre !static             // 5
      * ContractMarker+.*(..)) &&       // 6
    @annotation(pre) && this(obj);      // 7
```

Figure 6: PCD for Method Preconditions

Line 2 declares that two parameters will be bound, the annotation object, `pre`, which contains the test expression, and an object that implements the marker interface `ContractMarker`. This binding actually happens in line 7. The marker interface is injected into all types with the @Contract annotation (using a separate aspect), to make inheritance of tests easier to support; note the use of `ContractMarker+` in lines #4 and #7, to make sure that the join points in subclasses are matched. The `ContractMarker` object is the object under test.

Line 3 checks that preconditions tests are actually enabled, which can be configured globally through API calls and properties. (Postcondition and invariant tests can also be controlled this way.) Note that the preferred alternative for production deployments is to exclude the Contract4J aspects from the build, so that no DbC overhead is incurred at all. The referenced PCD in line 4 (not shown) is a typical PCD for excluding advising of the Contract4J code itself, to prevent infinite recursions, *etc.*

The key section of the PCD is in lines 5 and 6, highlighted in bold, where matching is done on execution join points of methods in `ContractMarker` and its subclasses. This PCD excludes constructors (handled separately) and only matches on nonstatic methods that have the `@Pre` annotation. Static methods are excluded because contracts focus on tests of instance state[5]. Note that since method annotations are not inherited in Java, we must require that the annotation appear on all method overrides[6]. If a subclass override does not have the same annotation, but the superclass implementation is invoked using `super…()`, the superclass method with the annotation will still be tested. However, even in this case Contract4J can't detect possible violations of the contract in the subclass method without the annotation and Contract4J can not currently detect that the annotation is missing.

Requiring the user to annotate all method overrides consistently is a design constraint reflecting a Java annotation limitation. However, even if method annotations where inherited, there is no way to write a pointcut that says "match a method in the class hierarchy if one of its ancestor methods has annotation A". Reflection could be used to handle this case (a possible enhancement), but it would be somewhat expensive to do.

An alternative would be to inject the missing annotation, if AspectJ's `declare parents` facility were generalized to support `declare method annotations`, for example, which could add an annotation to a method[7], assuming this is technically feasible. For this to be useful in the particular case discussed here, it would also be necessary for the declare statements to support a wider range of predicates, such as the pseudocode example suggested in Figure 7:

```
declare annotations:
  @Pre * ContractMarker+.method          // 1
```

[4] Some details have been altered for clarity and simplicity.

[5] However, you could argue that global (static) state could also be subject to testing. This may be supported in a future release.

[6] This was not a requirement for the original APT-based implementation, because the generated aspects no longer needed the annotations and would match on subclass overrides.

[7] Class annotations are already supported. Field annotation support is not needed in this case.

```
if (!@Pre * ContractMarker+.method  // 2
  && @Pre *ContractMarker.method)   // 3
```

Figure 7: "declare annotations" Extension

Here, the `@Pre` annotation is added to *`method`* in any subclass of `ContractMarker` (line 1) if it isn't already present (line 2), but it is present on the method in the top-level class or interface that defines the contract (line 3). How *`method`* is determined is intentionally left vague, but it would be the same method in all three lines. Note that Contract4J will locate the parent's test expression or generate a default expression, if no test expression is defined in a particular annotation.

At the very least, if this automatic mechanism can't be implemented (or the effort isn't otherwise justified), it would be useful if a mechanism exists to catch the user error of not annotating method overrides in subclasses.

In general, Contract4J's reliance on annotations points out some of the idiosyncrasies of Java 5 annotations, especially when used to represent a concept like DbC where expectations for inheritance behavior are different than for annotations.

3.2.2.2 Aspect for Field Invariant Tests When Fields Are Read or Written

Only invariant tests are supported for field reads and writes[8]. The lack of annotation inheritance that plagues method contract tests is not an issue here, since the field only "exists" in the class in which it is defined. Hence, if a field is annotated, all direct accesses will be correctly advised. However, field advice does have its own nuances.

3.2.2.2.1 Field "Gets"

Figure 8 shows the PCD for field "gets".

```
pointcut invarFieldGet (                 // 1
  Invar invar, ContractMarker obj):      // 2
    if (isInvarEnabled()) &&             // 3
    !within_c4j()  &&                    // 4
    !cflowbelow (execution               // 5
       (ContractMarker+.new(..))) &&     // 6
    get(@Invar * ContractMarker+.*) &&   // 7
    @annotation(invar) && target(obj);   // 8
```

Figure 8: PCD for Field Get Invariants

The first four lines are very similar to those for the method precondition PCD in Figure 6, with `Invar` substituted for `Pre`. In lines 5 and 6, I exclude field accesses that occur inside constructors, since we shouldn't expect the field to be initialized properly until the end of constructor execution. A separate aspect handles this special case. It uses the `percflow` instantiation model and matches on the initialization join points. Another aspect records accesses of any annotated fields and then after advice on the constructor evaluates the corresponding field tests after construction completes.

Because the field invariant test is evaluated at the end of construction, such a contract specification is not appropriate for a field that will be initialized on demand. In this case, a `@Post` test on the corresponding `get` method should be used.

[8] In principle, field pre- and postconditions could also be supported, but these tests are best added to bean property `get` and `set` methods, instead.

Back to Figure 8; note that the pointcut does not declare an Object argument for the returned field value, which could then be bound in an after `returning advice`, as shown in Figure 9.

```
after (                                  // 1
  Invar invar, ContractMarker obj)       // 2
  returning (Object result):             // 3
    invarFieldGet(invar, obj, result){   // 4
    …
}
```

Figure 9: Possible After Returning Advice

In fact, `around` advice is used for this and most other `@Post` test cases because of a special test feature supported by Contract4J, namely the ability to capture "old" values of context data, such as the value of the field before it is changed, so that the old and new values can be compared in some way[9]. I used this feature in the Figure 2 example to check that a withdrawal or deposit changed the account balance appropriately.

If the test expression specifies any "old" data, it is captured first in the around advice. Then, `proceed` is called to execute the join point and the value it returns is saved as the new field value.

3.2.2.2.2 Field "Sets"

Figure 10 shows the PCD for field "sets".

```
pointcut invarFieldSet (                 // 1
  Invar invar, ContractMarker obj,
      Object arg):                       // 2
    if (isInvarEnabled()) &&             // 3
    !within_c4j()  &&                    // 4
    !cflowbelow (execution               // 5
       (ContractMarker+.new(..))) &&     // 6
    set(@Invar * ContractMarker+.*) &&   // 7
    @annotation(invar) && target(obj)    // 8
      && args(arg);                      // 9
```

Figure 10: PCD for Field Get Invariants

The structure is very similar to the PCD in Figure 8 for field "gets", but now there is an extra `Object` parameter for the value being assigned to the field and of course `set(…)` join points replace `get(…)` join points.

Note that there is no way to actually bind an object to the field itself! Only the object being assigned to the field can be bound. Since Java variables are either references to objects or primitive values, this distinction is not important for Contract4j purposes, but it is possible that other applications using generic aspects may need to make this distinction. Perhaps AspectJ should support explicit binding to the field itself.

3.2.2.3 Advice

The advices used with these PCDs are all very similar. They use Java and AspectJ reflection APIs to fill in missing context information needed by the test expressions. They call support classes to create "default" test expressions when none is specified in the annotation. For invariant tests, they examine corresponding parent-class tests, if any, to ensure that the invariant tests are the same[10]. Finally, the advices call other support classes to package

[9] Only supported for primitives, Strings, and a few other classes.

[10] Only simple string comparison, ignoring white space, is currently supported, not true "semantic" equivalence. Hence "a==b" appears different from "b==a".

the information into the context structures required by Jexl and finally Jexl is invoked to execute the test. On failure, an error message is reported and program execution is stopped abruptly.

3.2.3 Contract4J "ContractBeans" Version

For completeness, I discuss the experimental *ContractBeans* (JavaBeans-like) version of Contract4J. The (PCDs) for this version are relatively simple, because most of the work must be done using reflection. Suppose I am testing the following class that uses the ContractBeans test approach.

```
class Foo (
  public int method(int i) {…}

  public boolean preMethod(int i) {…}

  public boolean
  postMethod(int result, int i) {…}
}
```

Figure 11: Foo Class Using *ContractBeans* Tests

Consider the precondition test case, where I could write a pointcut like the following.

```
pointcut pre(Foo foo, int i):
  call(int Foo.method(int)) &&
  hasMethod(boolean Foo.preMethod(int))
  && target(foo) && args(i);
```

Figure 12: Desired Pointcut for Precondition

The `hasMethod` pointcut specifier is a new undocumented experimental feature in AspectJ5 which tests for the existence of a method.

However, it is not possible to generalize this pointcut to arbitrary target classes and method signatures. It would require extending AspectJ to support a regular-expression syntax for matching strings, *e.g.*:

```
pointcut<T> pre(T t, Object[] args):
  call(* \(\T\)+.\(\M\)(\(\A\))) &&
  hasMethod(boolean $1.pre(cap($2)($3))
  && target(t) && args($3.values());
```

Figure 12: Possible Pointcut Regular Expression Syntax

In this contrived example, "\(…\)" indicates a capturing group, "\T" matches a type, "\M" matches a method name, "\A" matches the argument list, "$N" substitutes the value of the N^{th} capturing group, and "$3.values()" returns the list of values corresponding to the argument list captured by "$3"[11]. The made-up method "cap" handles capitalization of the method name, *i.e.,* conversion of the the first letter in the method name to upper case.

However, this syntax is hard to read and would therefore be error prone to use. Also, the merits of implementing regular expression support may not outweigh the effort required to implement it.

Instead, the ContractBeans version of Contract4J uses relatively simple, wide-reaching pointcuts and extensive runtime reflection to locate the test methods. First, end user is required to declare a "scoping" aspect that uses ITD to insert a marker interface into all classes where tests exist (or might exist), *e.g.*,

```
aspect scope (
  declare parents: (com.foo.bar..*)
    implements ContractMarker;
}
```

Figure 13: "Scoping" Aspect

Straightforward pointcuts are used to locate *all* possible join points where tests might be evaluated, within the defined scope. For example, the method precondition pointcut is shown in Figure 14.

```
pointcut preMethod (ContractMarker obj):
    if (isPreEnabled()) &&           // 3
    !within_c4j() &&                 // 4
    execution (!static               // 5
      * ContractMarker+.*(..)) &&    // 6
    this(obj);                       // 7
```

Figure 13: ContractBeans Pointcut for Method Preconditions

The key section of the PCD is lines 5 and 6, shown in bold. The rest of the PCD is similar to the boilerplate seen before. In fact, the whole PCD looks very similar to the annotation-based PCD for method preconditions shown in Figure 6, except that there are no annotations involved here. The annotation-based PCD will match only those join points where tests are actually defined, whereas the PCD in Figure 13 will match on *every* non-static method in the `com.foo.bar` hierarchy, adding significant overhead.

The corresponding advice uses reflection to determine if there is a `pre`*`Method`* test method to go with every method *`method`* found. The logic must look for methods with the appropriate name, that return boolean and that have a matching argument list, as discussed previously. The reflection adds a significant amount of overhead. Found methods are cached, but there is a non-trivial amount of setup effort required to determine the "key" for such a cache, so only modest performance gains are realized. In this case, it would help if AspectJ had a way of programmatically removing advice at the current join point, when a test method is not found by reflection, so all futile searches are never repeated.

3.2.4 User Adoption Issues

Because aspects can potentially affect the entire system, almost all aspect libraries include some mechanism for scoping the PCDs to only those packages and classes of interest. The following approaches are the most common.

- Define an abstract scoping pointcut in the library aspect and require the user to implement a concrete version of it that defines the packages and specific classes of interest. This minimizes, but does not eliminate the knowledge of AspectJ required by the user and the customization required to adopt a library.
- Define a marker interface that all library pointcuts use as a scoping construct, then require the user to "implement" or "extend" this interface in all classes or interfaces, respectively, where the user wants the aspect to apply. This is invasive if done manually. Instead, the user can write an aspect that uses intertype declaration to apply the interface where desired. (See, *e.g.,* Figure 13) This approach imposes about the same adoption effort and skill on the user as the scoping aspect option.
- Define an annotation that can be used instead of a marker interface (for Java 5 projects). Annotations can also be

[11] I said this was contrived!

introduced with ITD. Contract4J defines a `@Contract` annotation for this purpose. The curious thing about Contract4J usage is that the user will typically add this annotation manually, because the user will also need to add the other test annotations anyway in order to define tests. Hence, in practice, the user of Contract4J never needs to write any AspectJ code, although it will be necessary to introduce AspectJ into the build process.

- Define abstract base aspects and require the user to implement a derived aspect that implements abstract methods, supplies required callbacks, *etc.* A variation of this approach is to have concrete aspects use regular Java interfaces that the user must implement and "wire" to the aspect. This approach requires some user effort, but uses only familiar Java techniques.

The annotation form of Contract4J uses all these techniques internally. For example, classes with class-level "@Invar" tests get the marker interface `ContractMarker` through ITD, even though the annotations themselves are inherited. This apparent redundancy makes it easier to write PCDs that pick out the same join points on subclasses, even when they don't have the same "@Invar" annotation.

4. Conclusions

AspectJ's pointcut language enables succinct, yet powerful aspects when advice is needed at specific join points in known packages and classes. However, it is hard to write generic aspects that don't assume specific signature conventions, yet need details of the join points where they match in order to interact with the join points in non-trivial ways. Such aspects must use reflection to determine the additional information that they need.

Contract4J demonstrates the issues encountered when implementing a generic, reusable aspect library. In fact, it uses many of the "types" of PCDs you might expect to write, at least those focused on a single class, including field accesses, method calls, and instantiation, where specific coupling and computations are required for each case. Hence, developers of other generic library aspects are likely to encounter one or more of the same issues encountered in Contract4J. These issues will be a barrier to widespread development of rich AspectJ libraries unless some enhancements are made that simplify the issues involved.

Note that AspectJ 5 configuration files can be used to define some explicit name dependencies, thereby removing them from aspect code. However, this mechanism is insufficient for the needs of tools like Contract4J.

One possible solution is to extend the join point model with regular expression-like constructs, so that more sophisticated join point matching can be done on signature conventions without requiring explicit knowledge of "irrelevant" naming details. The aspect developer would then be able to bind more information through the PCD arguments for use in the advice bodies, thereby reducing the amount of reflection code required[12].

However, focusing on low-level constructs is probably the wrong enhancement strategy. Efforts to develop the theory and practice of aspect interfaces [7-8] are more important for the long-term evolution of AspectJ and AOSD in general. Components and aspects should be joined through interfaces that use the semantics of the concern, rather than being expressed through lower-level points of code execution, leading to the *concern semantics mismatch*.

Annotations that express meta-information about components are a first practical step in this direction. The Contract4J annotations form a design pattern that exposes key usage information about the component, in this case constraints on usage. Clients, including Contract4J aspects, IDEs, test generators, *etc.* interested in the "usage constaints concern" can work with the components in nontrivial ways through this "interface". However, even when matching on annotated join points in the Contract4J PCDs, the advice bodies still contain lots of low-level "plumbing", including calls to reflection APIs. Hence, annotations alone are not generally sufficient as an "aspect interface" to easily write powerful, yet generic aspects.

5. REFERENCES

[1] http://www.aspectj.org/

[2] http://barter.sourceforge.net/

[3] http://www.contract4j.org

[4] http://www.jcontract.org/

[5] http://jakarta.apache.org/commons/jexl/

[6] B. Meyer, *Object-Oriented Software Construction*, 2nd edition. Prentice Hall, Saddle River, NJ, 1997.

[7] G. Kiczales and M. Mezini, "Aspect-Oriented Programming and Modular Reasoning," *Proc. 27th Int'l Conf. Software Eng.* (ICSE 05), ACM Press, 2005, pp. 49-58.

[8] W. G. Griswold, *et al.,* "Modular Software Design with Crosscutting Interfaces", *IEEE Software*, vol. 23, no. 1, 2006, 51-60.

[9] Barbara Liskov, "Data Abstraction and Hierarchy," SIGPLAN Notices, vol. 23, no. 5, May, 1988.

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

[12] For most users, the relative runtime efficiencies of reflection *vs.* PCD binding, which may be similar, will be less important than the ease of development using either approach.

Java Virtual Machine support for Aspect-Oriented Programming

Alexandre Vasseur, Joakim Dahlstedt, BEA Systems
avasseur@bea.com, jda@bea.com
in affiliation with Jonas Bonér, Terracotta Inc.
jonas@terracottatech.com

BEA Systems, Java Runtime Product Group
Folkungagatan 122, S-102 65 Stockholm, Sweden

ABSTRACT

The majority of the frameworks for Aspect-Oriented Programming (AOP) use bytecode weaving, in addition to that, bytecode instrumentation is becoming more and more popular in enterprise software in general, as a way of adding services to applications, more or less transparently.

Unfortunately, there are many problems with bytecode instrumentation, problems that these frameworks and products will inherit. The key questions are: up to which point can bytecode instrumentation based weaving techniques scale and achieve manageability, transparency and efficiency? Is there a risk that products that are relying on these techniques will reach an end-point that limits further innovation towards more efficiency, ease of use and dynamicity?

We believe Java Virtual Machine (JVM) level support for AOP will address these issues and provide a solid ground for further innovations in the field of Aspect-Oriented Software Development (AOSD) in Java.

This report will first discuss the different implementation techniques for weaving, followed by a discussion of the problems with bytecode instrumentation based weaving that we have today, as well as potential future problems. This includes problems like, inefficient instrumentation, double bookkeeping, increasing complexity, multiple agents, reflective join points, etc.

We will then propose a novel technology for supporting AOP directly in the JVM that we have implemented in the JRockit JVM. A technology that we believe will address the problems outlined above.

Keywords

Aspect-Oriented Programming, AOP, AOSD, weaving, Java Virtual Machine, JVM

1. INTRODUCTION

Aspect-Oriented Programming [1] (AOP) is gaining momentum in the software community and the enterprise space at large. Introduced by Parc back in the 90's, it has been getting more and more mature through several initiatives and innovations in the research community, the open source community and the enterprise in the last two years. There has been a lot of traction in the Java community that recently lead to the merger of AspectWerkz [3] and AspectJ [4] now housed at the Eclipse under the name AspectJ 5 [5]. AspectJ is sponsored by BEA Systems and IBM and can be considered as the de-facto standard for AOP in Java.

As the popularity of AOP is growing and the research community is moving things forward, vocabulary, concepts, and implementations have gained in consistency, allowing for better tool support and developer experience - such as with AspectJ Eclipse plug-in AspectJ Development Tools (AJDT) [6].

AOP has gone through several implementations techniques, ranging from source code weaving to bytecode instrumentation based weaving. Bytecode instrumentation is the technique that has been most widely adopted in Java, in particular after the advent of Java 5 JVMTI [7]. Bytecode instrumentation is now used by several enterprise products in the area of application management and monitoring and more recently Plain Old Java Object (POJO) based middleware [13] and transparent clustering [14].

Unfortunately, there are many problems with bytecode instrumentation, problems that these frameworks and products will inherit. The key questions are: up to which point can bytecode instrumentation based weaving techniques scale and achieve manageability, transparency and efficiency? Is there a risk that products that are relying on these techniques will reach an end-point that limits further innovation towards more efficiency, ease of use and dynamicity?

Up to which point can bytecode instrumentation based weaving techniques scale and achieve manageability, transparency and efficiency? Is there a risk that AOP implementations that are relying on these techniques will reach an end-point that limits further innovation towards more efficiency, ease of use and dynamicity?

We believe Java Virtual Machine (JVM) level support for AOP will address these issues and provide a solid ground for further

innovations in the field of Aspect-Oriented Software Development (AOSD) in Java.

This report will first discuss the different implementation techniques for weaving, followed by a discussion of the problems with bytecode instrumentation based weaving that we have today, as well as potential future problems. This includes problems like, inefficient instrumentation, double bookkeeping, increasing complexity, multiple agents, reflective join points, etc.

We will then propose a novel technology for supporting AOP directly in the JVM that we have implemented in the JRockit JVM [8]. A technology that we believe will address to the problems outlined above.

2. CURRENT STATE OF WEAVING

Weaving is the process of taking cross-cutting code and the regular "base" application and "weave" them into one single unit, one single application.

Weaving can happen at different periods in time:

- Compile time weaving: post processing the code, e.g. ahead of deployment time (thus ahead of runtime) (as in AspectJ 1.x).
- Load time weaving: weaving is done as the classes are loaded i.e. at deployment time (as in AspectWerkz 2.0).
- Runtime weaving: weaving can occur at any time during the lifetime of the application (as in JRockit and the SteamLoom Project [12]).

This process can also be done in many different ways:

- Source code weaving: input is the developed source code and output is modified source code that invokes the aspects (as in AspectJ 1.0).
- Bytecode weaving: input is the compiled application classes' bytecode and output is modified bytecode of the woven application (as in AspectWerkz 2.0 and AspectJ 1.1 and beyond).

Source code weaving is limited in the sense that all source code must be available and presented to the weaver so that aspects can be applied. This makes it impossible for example to implement generic monitoring services - with or without use of explicit AOP constructs. Compile time weaving suffers from the same problem. All bytecode that will be deployed needs to be prepared before deployment in a post compile time.

Bytecode weaving vs. JVM weaving is the subject of this report and will be discussed in the following sections.

A side note is that a limited form of weaving has been available in the JVM for some time: dynamic proxies [15]. This API has been part of the JDK since 1.3 and it allows to create a dynamic proxy of an interface (or a set of interfaces) which gives the possibility of intercepting each invocation to the interface(s) declared methods and redirect it to an invocation handler implementing arbitrary logic. This is not really weaving by definition, but it resembles it, in the way that it gives a simple way of doing method interception. This technique is used by various frameworks to do simple form of AOP, for example, the Spring Framework [16].

3. PROBLEMS WITH BYTECODE INSTRUMENTATION BASED WEAVING

It is worth emphasizing that the problems described below are tied to bytecode instrumentation and as a consequence affects different AOP implementations (such as AspectJ). These problems have impact on all bytecode instrumentation based products in general, such as application monitoring solutions, profiling tools or other AOP-applied solutions as their use is getting more and more popular.

3.1 Instrumentation is inefficient

The actual instrumentation part of the weaving is usually very CPU intensive, and sometimes also consumes significant amounts of memory. This can affect startup time. For example, to intercept all calls to the `toString():String` method or all accesses to a certain field, one needs to parse almost every single bytecode instruction in all classes, one by one. This also means that a lot of intermediate representation structures will be created by the bytecode instrumentation framework to expose the bytecode instructions in a usable way. This could potentially mean that the weaver needs to parse all bytecode instructions in all classes in the whole application (including third-party libraries etc.), e.g. in the worst case more than 10,000 classes.

If more than one weaver is use, the overhead will be multiplied.

For some pointcut expressions, the underlying weaver can be optimized by first looking at the class' bytecode constant pool section to determine if there might be a match. Unfortunately this approach can not be used for most of the pointcut for which the complete class hierarchy and member database must be known in order to perform the matching – as discussed below.

3.2 Double bookkeeping: Building a class database for the weaver is expensive

In order to know whether a class/method/field should be weaved or not, the weaver needs to do matching on metadata for this class or member. Most AOP frameworks and AOP-applied products have some sort of high level expression language (pointcut expressions) to define where (at which join points) a code block (advice) should be weaved in. These expression languages can for example let you pick out all methods that have a return type which implements an interface of type T. This information is not available in the bytecode instructions representing the call to a specific method M. The only way of knowing if this specific method M should be weaved or not is to look it up in some sort of class database, query its return type and check if its return type implements the given interface T.

You might be thinking: why not just use the `java.lang.reflect` API? The problem with using reflection here is that there is no way of querying a Java type reflectively without triggering the class loading of this particular class, which will trigger the weaving of this class before we know enough about it in order to do the weaving (in load time weaving infrastructures). Simply put: we end up with the classic chicken and egg problem.

The weaver therefore needs a class database (usually built up in memory from the raw bytecode read from the disk) to do the required queries on if needed for the actual join points to be

found. This problem can sometimes be avoided by limiting the expression language expressiveness, but that usually limits the usability of the product.

This in memory class database is also redundant once the weaving is done. The JVM already has all this information in its own database, well optimized, (that for example serves the `java.lang.reflect` API). So we end up doing double bookkeeping of the whole class structure (object model) which consumes significant and unnecessary memory, as well as adds a startup cost in creating this class database, and maintaining it when a change occurs.

If more than one weaver is use, the overhead will be multiplied – as in most cases each weaver maintains its own class database.

3.3 Changing bytecode at runtime adds more complexity

Java 5 brought the HotSwap API [9] as part of the JVMTI specification. Before Java 5, this API was only available when running in debug mode, and only for native C/C++ JVM extensions. It allows changing the bytecode, i.e. to redefine a class, at runtime. It is used by some AOP frameworks, and AOP-applied products to emulate runtime weaving capabilities.

Despite being very powerful, this API limits usability, scalability and is inefficient. Since bytecode is being changed at runtime, instrumentation costs (CPU overhead and memory overhead) are also happening at runtime. Also, if there is a need to do a change in many places, this means redefining many classes as well. The JVM will then have to redo all the optimization and inlinings that it may have done.

It is also very limited. The API does not specify where the current running bytecode can be retrieved. A weaver thus needs to make the assumption that this bytecode is on disk, or it needs to keep track of it. This is a major issue when multiple weavers are used as explained in the next section.

Further, none of the current implementations of the HotSwap API supports schema change which the specification states as being optional. This means that it is not possible to change the schema of a class at runtime, e.g. add methods/fields/interfaces etc that might be needed for the underlying instrumentation model. This makes it impossible to implement certain types of runtime weaving and thus requires the user to "prepare" the classes in advance. Such a technique is for example used in AspectWerkz[3] to provide hot-deployment of around advice.

3.4 Multiple agents is a problem

When multiple products are using bytecode instrumentation, unexpected problems may happen. Problems related to precedence, notification of changes, undoing of changes etc. This perhaps is not so much a problem today, but this will be a significant problem in the future. A weaver can be seen as an agent (as referred to in the JVMTI specification) that performs instrumentation at load time or runtime. When there are multiple agents, it is a high risk that the agents will get in each others way, changing the bytecode in a way that was not expected by the next agent, making the assumption that it is the sole configured agent.

Here is an example of a problem that can happen when two agents are unaware of each other. If for example an application uses two agents, one AOP weaver and one application performance product (that are both doing bytecode instrumentation at load time), there is a risk that the woven code may not be part of the performance measurement as illustrated below:

```
// Say this is the original user code
void businessMethod() {
    userCode.do();
}

// -- Case 1
// Say the AOP weaver was applied BEFORE the
// performance management weaver
// The weaved code will behave like:
void businessMethod() {
    try {
        performanceEnter();
        // hypothetical advice
        aopBeforeExecuting();
        userCode.do();
    } finally {
        performanceExit();
    }
}
// i.e. the AOP code will affect the measure

// -- Case 2
// Say the AOP weaver was applied AFTER the
// performance management weaver
// The weaved code will behave like:
void businessMethod() {
    //hypothetical advice
    aopBeforeExecuting();
    try {
        performanceEnter();
        userCode.do();
    } finally {
        performanceExit();
    }
}
// i.e. the AOP code will NOT affect the measure
```

This illustrates a problem with precedence between the agent: there is no fine grained configuration to control the ordering at a join point (or pointcut) level. The ordering is not well-defined.

Some other situations might lead to more unpredictable results. For example when a field access is intercepted, it usually means that the field get bytecode instructions are moved to a newly added method and replaced by a call to this new method. The next weaver will thus see a field access from another place in the code (from this newly added method) that then might not be matched by its own matching mechanism and configuration.

To summarize, the main problems are:

Which bytecode does the agent see? The problem is that normally the bytecode to be weaved is obtained from the class loading pipeline but the dependent bytecode to build up the class database from is read from disk. When multiple agents are involved bytecode on disk is not anymore the one being executed, since some agent might have already changed the bytecode. This means that the second agent has an incorrect view of the bytecode. This also happens when the HotSwap API is used.

3.5 Intercepting reflective calls is tedious

Current weaving approaches can only instrument execution flows that can be (at least partially) statically determined. Consider the following code sample that invokes the method `void doA()` on the given instance `foo`:

```
public void invokeA(Object foo) throws Throwable {
    Method mA = foo.getClass().getDeclaredMethod(
        "doA", new Class[0]
    );
    mA.invoke(foo, new Object[0]);
}
```

This kind of reflective access is often used in modern libraries, to create instances, to invoke methods, or to access fields.

From a bytecode perspective, the call to the method `void doA()` is not seen. The weaver will only see calls to java.lang.reflect API. There is no simple and performant way of weaving calls that are made reflectively. This is an important limitation in how weaving can be done and how AOP is implemented today. Best practices recommend the developer to use execution side pointcuts instead. Obviously, from a JVM perspective, there will be a method dispatch to the `doA()` method, even if it does not appear in the source code or bytecode. JVM weaving has proven to be to be the only weaving mechanism that addresses this issue in an efficient way.

3.6 Other problems

Bytecode instrumentation, especially when done on-the-fly (load time or runtime), is seen with skepticism by some people. There is an emotional angle to changing code on-the-fly that should not be underestimated, especially when it is paired up with a mind-bending revolutionary new technology such as AOP or transparent injection of services. Clashes that may happen when multiple agents are involved will increase this skepticism.

Another potential problem is the 64K boundary for class files stated in the Java specification. Method bodies are limited to a 64K total bytecode instruction size. This might be a problem when weaving already large class files, like for example the resulting class file when compiling a JavaServer Pages (JSP) [17] file to a Servlet [18]. When instrumenting this class, it might break the 64K limit and then cause a runtime error.

4. PROPOSED SOLUTION

4.1 JVM support for weaving

JVM weaving is the natural answer to most of the issues discussed above. The following examples show that the JVM is already doing most of the work involved to do weaving: When a class gets loaded, the JVM does read the bytecode to build up the data needed to serve the `java.lang.reflect.*` API purpose. Another example is method dispatching. Modern JVMs compile the bytecode of methods or code blocks to more advanced and efficient constructs and execution flows (doing code inlining where applicable). Due to the HotSwap API requirements, the JRockit JVM (and probably other JVMs too) also bookkeeps which method calls which other method, so that a method body can still be hotswapped in all expected places - inlined or not - if its defining class is redefined at runtime.

As a consequence, instead of changing the bytecode to weave in an advice invocation - say before a specific method call - the JVM could actually have knowledge about it and simply do a dispatch to the advice at any matching join point prior dispatching to the actual method.

As bytecode would be untouched, immediate advantages can be expected such as

- no startup cost due to bytecode instrumentation
- full runtime support to add and remove advices at any place, any time, at linear cost
- implicit support to advise reflective invocations
- no extra memory consumption to replicate the class model to some framework specific structures

This is very different from the C level events that have been defined in the JVMDI specification [10], such as JVMDI_EVENT_METHOD_ENTRY or JVMDI_EVENT_FIELD_ACCESS. In the JVMDI case, first one would have to deal with C level API, which makes it complex for most developers and fragile or complex to distribute, and second the specification does not provide a fine grained join point matching mechanism but actually requires ones to subscribe to all such events, thus still happening with an undeniable overhead - hence the D(ebug) in "JVMDI".

The following code sample introduces the JRockit weaving API that will be detailed in the next section.. The program below dispatches to the static method `advice()` just before the `sayHello()` method gets called :

```
public class Hello {
    // -- The sample method to intercept
    public void sayHello() {
        System.out.println("Hello World!");
    }

    // -- Using the JRockit JVM support for AOP
    static void weave() throws Throwable {
        // match on method name
        StringFilter methodName =
            new StringFilter(
                "sayHello",
                StringFilter.Type.EXACT
            );

        // match on callee type
        ClassFilter klass = new ClassFilter(
            Hello.class,
            false,
```

```
            null
        );
        // advice is a regular method dispatch
    Method advice =
        Aspect.class.getDeclaredMethod(
            "advice",
            new Class[0]
        );

    // Get a JRockit weaver and subscribe
    // the advice to the join point picked
    // out by the filter
    Weaver w = WeaverFactory.createWeaver();

    w.addSubscription(new MethodSubscription(
        new MethodFilter(
            0,
            null,
            klass,
            methodName,
            null,
            null
        ),
        MethodSubscription.InsertionType.BEFORE,
        advice
        ));
    }

    // -- Sample code
    static void test() {
        new Hello().sayHello();
    }

    public static void main(String a[])
        throws Throwable {
        weave();
        test();
    }

    // -- Sample aspect
    public static class Aspect {
        public static void advice() {
            System.out.println("About to say: ");
        }
    }
}
```

4.2 Subscription and action based model

The JRockit JVM AOP support exposes a Java API that is deeply integrated in the JVM method dispatching and object model components. In order to not tie the JVM to any current or future AOP specific technology direction we have decided to implement an action dispatch and subscription model.

The API allows defining well defined subscriptions at specified pointcuts, for which it is possible to register one action to which the JVM will dispatch. An action is composed of

- a regular java method - that we will reference as the action method - that will be invoked for each join point that matches the subscription
- an optional action instance on which to invoke the action method
- an optional set of parameter-level annotations that dictate the JVM which arguments the action method is expecting from the invocation stack.

The action can be flagged as a before action, an after returning action, an after throwing action or an instead-of action (similar to AOP around concept).

In order to invoke the API, one has to get a handle to a `jrockit.ext.weaving.Weaver` instance. The weaver instance will control which operations are allowed according to its caller context. For example you might not want a deployed application in an application server to create a weaver to subscribe action methods to some container level or JDK specific join points, while a container level weaver may actually subscribe to application specific join points. This weaver visibility concept mirrors the visibility rules from the underlying class loaders delegation model.

A very simple comparison of how these constructs are mapped to the regular AOP constructs might shed some light on this model:

- the subscription can be seen as a kinded pointcut, or actually a kinded pointcut (field *get(),set(),* method *call()*, etc) composed with a *within()/withincode()* pointcut.
- the action instance can be seen as the aspect instance
- the action method can be seen as the advice

Readers familiar with AOP may already understand that to implement a complete AOP framework on top this JVM level API, some more development will be required. An intermediate layer which will manage the aspect instantiation models (per clause), implement the *cflow()* pointcuts, and implement full pointcut composition and orthogonality is required.

4.3 The action method

An action method (similar to the AOP advice concept) is as a regular Java method of a regular class (that will act as the aspect). It can either be static method or member method. Its return type has to follow some implicit conventions. Its return type should be void for a before action, and should be of the type that will be placed on the stack as the result of the action invocation for an instead-of action (similar to AOP around advice semantics).

The action method can have parameters, whose annotations further control context exposure as illustrated in the following code sample:

```
public class SimpleAction {
   public static void simpleStaticAction() {
       print("hello static action!");
```

```
    }
    public void simpleAction() {
        print("hello action!");
    }
    public void simpleAction(
             @CalleeMethod WMethod calleeM,
             @CallerMethod WMethod callerM) {
         print(callerM.getMethod().getName());
         print(" calling ");
         print(calleeM.getMethod().getName());
     }
}
```

This code sample introduces the `jrockit.ext.weaving.WMethod`. This method acts as a wrapper for `java.lang.reflect.Method`, `java.lang.reflect.Constructor` and the class' static initializer which is not represented in java.lang.reflect.*. This is similar to the AspectJ `JoinPoint.StaticPart.getSignature()` abstraction.

In order to support instead-of and the ability to decide whether to proceed the interception chain or not (as implemented in AOP through the concept of `JoinPoint.proceed()`) we also introduced the `jrockit.ext.weaving.InvocationContext` construct as illustrated below.

```
public class InsteadOfAction {
    public Object instead(
            InvocationContext jp,
            @CalleeMethod WMethod calleeM) {
        return jp.proceed();
    }
}
```

4.4 The action instance and the action kind

As illustrated in the previous code samples, an action method can be either static or not. If the action method is not static, ones need to pass in an action instance on which the JVM will invoke the action method.

This follows the syntax style with which a Java developer would invoke a method reflectively using `java.lang.reflect.Method.invoke(null/*static method*/, .../*args*/)`. Although, as opposed to this example, with JVM AOP support, the underlying action invocation will not involve any reflection at all.

Giving the user control over the action instance opens up many interesting use cases. For example, one could implement a simple delegation pattern to swap a whole action instance with a different implementation at runtime, without having to involve the JVM internals.

For addition, this will be useful to implement AOP aspect instantiation models (per clause) such as *issingleton(), pertarget(), perthis(), percflow()* etc, while not locking the JVM API to some predefined semantics.

Before registering the subscription to the weaver instance, it is given a kind that acts as the advice kind: *before, instead-of, after-returning* or *after-throwing*.

The code to create a subscription is as follow:

```
// Get a Weaver instance that will acts a
// container for the subscription(s) we create
Weaver w = WeaverFactory.getWeaver();

// Regular java.lang.reflect is used to refer
// to the action method "simpleStaticAction()"
Method staticActionMethod =
    SimpleAction.class.getDeclaredMethod(
        "simpleStaticAction",
        new Class[0]//no arguments
    );

MethodSubscription ms = new MethodSubscription(
    .../* where to match*/,
    InsertionType.BEFORE,
    staticActionMethod
);
w.addSubscription(ms);

// Use of an action instance to refer to the
// non static action method "simpleAction()"
Method actionMethod =
    SimpleAction.class.getDeclaredMethod(
        "simpleAction",
        new Class[0]// no arguments
    );

// Instantiate the action instance
SimpleAction action = new SimpleAction();
MethodSubscription ms2 = new MethodSubscription(
    ...,// where to match, explained below
    InsertionType.BEFORE,
    actionMethod,
    actionInstance
);
w.addSubscription(ms2);
```

AOP semantics such as *within()* and *withincode()* type patterns are also implemented through variations around this API.

4.5 Subscription on events

As shown in the previous code sample, the subscription API is relying on the `java.lang.reflect.*` object model and some simple abstraction like `jrockit.ext.weaving.WMethod` to unify Method, Constructor and class' static initializer handling.

Subscriptions can be made on events triggered by: field access and modification, method and constructor calls, exception throwing and catching as well as static initialization of a class.

If we, for example, look at a method subscription construct, the first parameter in the call to `new jrockit.ext.weaving.MethodSubscription(...)`, must be a `jrockit.ext.weaving.Filter` instance, which has several concrete implementations, to match on methods, fields etc.

A `jrockit.ext.weaving.MethodFilter` instance will act as the definition on which the JVM weaver implementation will do the join point shadow matching for method and constructor call pointcuts. A `MethodFilter` allows filtering on modifiers, annotations, declaring type, name, return type (it also exposes extra structures to support within()/withincode() semantics):

The user can also pass in a `jrockit.ext.weaving.UserDefinedFilter` instance to implement a finer matching logic. The `UserDefinedFilter` callback mechanism is used to implement more advanced matching schemes (a concept that is similar to Spring AOP's `MethodMatcher` and `ClassFilter`).

All of these structures are optional, and if null is encountered, it means "match any".

The following will thus match all method calls whose name starts with "bar". Note that we pass in several null values in this very simple case:

```
StringFilter sf = new StringFilter(
    "bar", STARTSWITH
);

MethodFilter mf = new MethodFilter(
    0, null, null, sf, null, null
);

MethodSubscription ms = new MethodSubscription(
    mf,
    InsertionType.BEFORE,
    staticActionMethod
);
w.addSubscription(ms);
```

5. DISCUSSION

5.1 Benefits

There are several benefits in using JVM weaving instead of bytecode instrumentation. From an high level perspective, the weaving appears as a natural extension to the JVM capabilities. It is less intrusive in many ways with performance, scalability and usability benefits.

5.1.1 No more bytecode instrumentation increases scalability

The bytecode is not modified. The regular compilation pipeline from bytecode to executable code is followed in the JVM internals. There is no more need to parse the bytecode instructions and represent them in some intermediate structures.

The weaver becomes ubiquitous. Even though ones may want to register subscriptions at startup time, it is no longer a requirement. This greatly reduces the startup time of an application since there is no need at all to analyze bytecode instructions in order to find the interesting join points. This also gives the opportunity to develop truly dynamic systems, allowing deployment and undeployment of aspects at any point in time without any extra overhead or complexity.

5.1.2 No more redundant bookkeeping of types decreases memory usage and improves scalability

As bytecode instrumentation does not occur anymore, there is no longer a problem with double bookkeeping the object model. The subscription API relies on the `java.lang.reflect.*` model that already provides this information in a familiar way to the Java developer.

5.1.3 Multiple agents are kept consistent

As the bytecode of the woven classes does not get modified, there is no more risk of conflicts between two different agents changing the bytecode in two different ways that might be incompatible - hiding properties of the original program from another. The registration order of the subscription acts as the precedence rule. If a class is marked as being `Serializable`, it means that no hidden structures needed to support the runtime execution of the woven advice is added, which means that regular serialization will be fully supported. Usually bytecode instrumentation techniques need to ensure that serialization is preserved (for example dealing with the `serialVersionUID` field).

5.1.4 Support for intercepting reflective invocations

By using JVM level method dispatching, all reflective calls (method invocation or field get or set) can be matched as if they were regular calls and all registered actions will then get triggered. This occurs without any extra cost or implementation specific details and complexity.

5.2 Limitations

Some fine-grained semantics that AspectJ defines are not easily addressed at the JVM level. AspectJ for example supports the concept of *preinitialization*, *initialization* and *constructor execution* pointcuts. A constructor execution pointcut will pick out the constructor as it appears in the source code, while the initialization pointcut will pick out all constructor execution(s) that lead to having an initialized instance, thus including this(...) constructor delegations. Such a difference is not easily handled by the JVM. It may actually also be compiler dependant where more aggressive inlining strategies may occur. Consider for example:

```
public class Timeout {
    int delay;
    String cause;
    Timeout(int aDelay, String aCause) {
        this.delay = aDelay;
        this.aCause = aCause;
    }
    Timeout(int aDelay) {
        this(aDelay, "unknown");
        // A compiler could produce
        // inlined equivalent:
        // -- no call to this(..,..) but instead:
        // this.delay = aDelay;
```

```
        // this.aCause = "unknown";
    }
}
```

Since we currently do not have a grammar (AST etc.) for defining pointcut expressions, but a Java API, supporting a very expressive and fine-grained pointcut language like for example the one in AspectJ, will require an additional abstraction layer on top of the existing API. Part of it is already available as the pointcut parsing and matching logic based on `java.lang.reflect.*` structures has been introduced in recent AspectJ 5 releases and is now accessible in the `org.aspectj.weaver.tools` package.

6. CONCLUSIONS

Bytecode instrumentation techniques are now widely used in the Java platform in several different areas, ranging from Aspect-Oriented Software Development to more specific applied solutions such as application monitoring, persistence, or distributed computing. In an attempt to become more usable and transparent, the techniques of load time weaving and instrumentation at deployment time is becoming popular.

Unfortunately, such techniques do not provide the proper properties to match scalability and usability requirements, especially as it becomes more and more used, and mixed through the use of several different instrumenting agents from different products. JVM weaving and JVM support for AOP as implemented in JRockit happen to be a natural way approach to the problem and drive the innovation and the technology further. The proposed Java API that bridges JVM method dispatching internals to user defined action and subscription depending solely on the `java.lang.reflect` API fills the gap elegantly, as well as addressing major scalability and usability issues.

Nevertheless, widespread adoption requires a good assessment of this new API towards real use cases - such as AOP or runtime adaptability of large applications.

7. FUTURE WORK

Despite that JVM weaving brings huge value and addresses scalability and usability problems tied to bytecode instrumentation techniques, there are still some interesting drawbacks that will need to be solved so that some use cases can be addressed completely - possibly with complimentary approaches.

Some bytecode instrumentation based products are using very fine grained change that may not be possible to mirror in the (current) JVM AOP API. There are for example some use cases that deal with the synchronized blocks, so that different locking strategies - for example distributed - can be transparently injected into a regular application. Such a fine grained manipulation usually requires conditional execution of the synchronized block, or even complete removal of it so that it is replaced by some proprietary locking API call. Such a specific need can be addressed within the JVM but it is actually impossible to come with a solution that works for each use-case and that is efficient. It is also interesting to note that leading AOP frameworks are not (yet) exposing the synchronized block (or monitor entry and monitor exit lock acquisition and release) as join points.

Native support for various action instance life-cycles is something that would be good to have. For example support for pure per instance based deployments of action instances, similar to the work done in SteamLoom [12] is interesting.

As bytecode instrumentation is gaining popularity, introducing such a new API is not neutral. It would represent a fairly high cost to have a product developed so that it works for JVM that would support this API - such as JRockit - and for JVM that would not support this API. A specification, for example a Java Community Process (JCP) [19], in this area would be beneficial.

8. REFERENCES

[1] Kiczales, G., Lamping, J., Mendhekar, A. ,Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J., Aspect-Oriented Programming, ECOOP1997

[2] Gosling, J., Joy, B., Steele, G. The Java Language Specification (2nd edition) Addison-Wesley, 2000.

[3] Bonér,J., Vasseur,A., AspectWerkz, a dynamic, lightweight and high-performant AOP framework for Java, 2002-2005. (http://aspectwerkz.codehaus.org)

[4] Kiczales,G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W. G., An Overview of AspectJ, ECOOP 2001.

[5] Press release, January 19th, 2005, "AspectJ and AspectWerkz to Join Forces" (http://www.eclipse.org/aspectj/aj5announce.html)

[6] AspectJ Development Tools – AJDT (http://www.eclipse.org/ajdt/)

[7] JVMTI (JSR-163 - http://www.jcp.org/en/jsr/detail?id=163)

[8] JRockit JVM, BEA Systems (http://www.jrockit.com)

[9] HotSwap API (JSR-163 - http://www.jcp.org/en/jsr/detail?id=163)

[10] JVMDI, Java Virtual Machine Debug Interface (http://java.sun.com/j2se/1.5.0/docs/guide/jpda/jvmdi-spec.html)

[11] JSR-175, A Metadata Facility for the Java Programming Language (http://www.jcp.org/en/jsr/detail?id=175)

[12] Bockisch, C., Haupt, M., Mezini, M., Ostermann, K.: Virtual Machine Support for Dynamic Join Points. In: International Conference on Aspect-Oriented Software Development. (2004)

[13] JBoss EJB 3 (http://www.jboss.org)

[14] Terracotta Virtualization Server (http://www.terracottatech.com)

[15] Java Dynamic Proxy Classes (http://java.sun.com/j2se/1.5.0/docs/guide/reflection/proxy.html)

[16] The Spring Framework (http://www.springframework.org)

[17] JavaServer Pages Technology (JSP) (http://java.sun.com/products/jsp)

[18] Java Servlet Technology (http://java.sun.com/products/servlet/)

[19] Java Community Process (JCP, http://jcp.org)

Lessons learned building tool support for AspectJ

Mik Kersten, University of British Columbia, beatmik@acm.org
Matt Chapman, IBM United Kingdom Ltd., mchapman@uk.ibm.com
Andy Clement, IBM United Kingdom Ltd., clemas@uk.ibm.com
Adrian Colyer, Interface21 Ltd., adrian.colyer@interface21.com

ABSTRACT
A key part of the AspectJ technology is the integrated development environment integration, which makes it possible for developers to use aspect-oriented programming without giving up the tool support to which they are accustomed. In this experience report we summarize our current tool suite, and discuss the lessons we learned extending object-oriented development environments to make crosscutting structure explicit. We also report on the challenges of surfacing aspects in structure views without encouraging misconceptions about the language, our successes and failures in extending object-oriented tools, and our ongoing work in supporting advanced IDE features and exposing the dynamic properties of the language.

Keywords
Aspect-oriented programming, integrated development environments, language design and implementation, software tools

1. INTRODUCTION

Object-oriented programming (OOP) tool support makes the inheritance and encapsulation structure of a system explicit. Many programmers rely on query facilities to find overriding methods, tree views to inspect the system's type hierarchy, or debugger mappings from exception traces to offending source lines. AspectJ [10] is an Aspect-Oriented Programming (AOP) [11] extension to Java [7] that provides the programmer with language support for modularizing crosscutting. The goal of the AspectJ tool support is to make the crosscutting structure of the system explicit.

The AspectJ IDE plug-ins have been crucial to the adoption of the AspectJ language for both practical and pedagogical reasons. They allow programmers to use AspectJ without forcing them to give up the other development tools they already use. Subtle but fundamental assumptions in the object-oriented IDEs' structure models and views made it challenging to implement our AOP extensions to these tools. In this experience paper we report the lessons we learned from integrating aspects into existing IDE's models and views. We also discuss the technical issues that we observed while building facilities for editing, compiling, navigating, documenting, and debugging aspect-oriented programs. Section 2 provides an overview of our current tool suite. Sections 3 & 4 discuss what we learned designing and implementing these tools.

2. THE ASPECTJ TOOL SUITE
The first tool we released was a command line compiler. As our user community grew the number of requests for build and development environment integration increased. Users wanted the ability to invoke the AspectJ compiler from within their development environment [9]. In addition, they asked IDE integration that provided the editing and navigation facilities that they were accustomed to in their Java IDEs. In 1999 we set out to design and integrate IDE facilities for working with crosscutting structure.

The first step we took was to show crosscutting structure as navigable textual annotations in Emacs[1], and as links in Javadoc [6] documentation generated with the ajdoc tool. We met the increasing user demand with plug-ins for the JBuilder[2], NetBeans[3], and Eclipse[4] IDEs. Since we were not able to support all IDEs used by our community, we also provided a standalone tool called the AJBrowser for navigating aspects, and an Ant[5] task for integrating the AspectJ compiler into an existing build process. Figure 1 provides a historical perspective on the tool support we released as the language matured and demand grew. Intervals on the timeline indicate periods of active contribution to those tools. Section 4.4 discusses why some tool lasted while others did not.

2.1 AspectJ
AOP enables the modular implementation of crosscutting concerns. Such concerns are inherent in complex systems, and are impossible to capture cleanly with OOP. AspectJ is an extension to Java that provides the programmer with language mechanisms that explicitly capture crosscutting structure. The result is similar to the benefits of OOP modularity for object encapsulation and inheritance: easier to maintain code with greater potential for reuse.

---

[1] http://www.gnu.org/software/emacs

[2] http://www.borland.com/jbuilder

[3] http://netbeans.org

[4] http://eclipse.org

[5] http://ant.apache.org

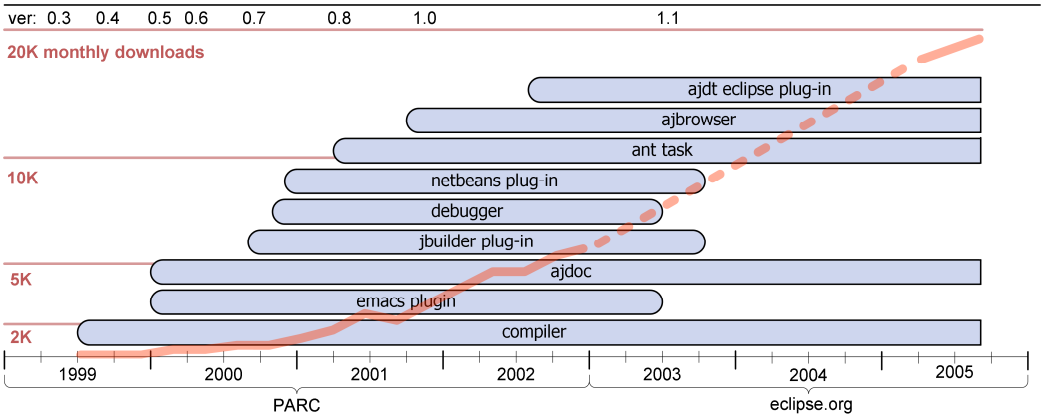


Figure 1. Timeline of AspectJ tool releases (dashed line indicates missing statistics)

Consider the failure handling policy in Figure 2. This simple aspect declares that after a `WSIFException` is thrown in any public method within the `org.apache.wsif` package the exception will be handled by the body of the advice (line 7). The aspect defines a `pointcut` as the set of join points corresponding to every execution of a public method within that package. In AspectJ's pointcut and advice mechanism, join points are key points in the dynamic execution of the program [13]. These include method and construction execution, class initialization, and field accessors. The `FailurePolicyEnforcement` aspect uses `after` advice to define what action should be taken under these points in the execution. Advice execution can also be specified to happen `before` and `around` a join point. Aspects as simple as this one have been demonstrated to be expressive enough to capture crosscutting concerns such as first failure data capture policies in application servers [2].

OOP implementations corresponding to Figure 2 result in the exception handling code being scattered and tangled across the system. By making these concerns explicit, AspectJ provides the expected benefits of good modularity for crosscutting concerns. The goal of the AspectJ tool support is to show the structure of well-modularized crosscutting concerns.

2.2 Crosscutting structure views

The most common objection that arose during our tutorials and presentations suggested that AspectJ's ability to introduce behavior invoked implicitly would compromise program understandability. For example, `around` advice can prevent a method from executing. Following references or imports from the containing file does not help the programmer figure out when this might happen, since aspects crosscut the type structure. Without tool support the only way to know how aspects affect a particular method is to examine all of the aspects. To address this, the AspectJ tools make crosscutting structure explicit by indicating which advice will affect that method. For example crosscutting links in the Cross References view make it possible to navigate from a method signature to advice that may affect the execution of that method (Figure 2, right hand side). These links are also exposed as inline annotations in the editor ruler (Figure 2, left hand edge). Right-clicking on the annotations displays a context menu with the same relationships. The relationships are shown in both directions, to allow consistent navigation between the source and target of advice.

AspectJ's inter-type declarations, an open class mechanism that originated from Flavors [12], affect the type structure of the system. For example, a class may have new members declared on it. This structure is made explicit in the Cross References view as well (Figure 3). Unlike Java member declarations, advice and some inter-type declarations cannot be invoked explicitly. As such, these declarations are not named. In the Cross References view they are distinguished by their kinds and by the pointcuts that they reference.

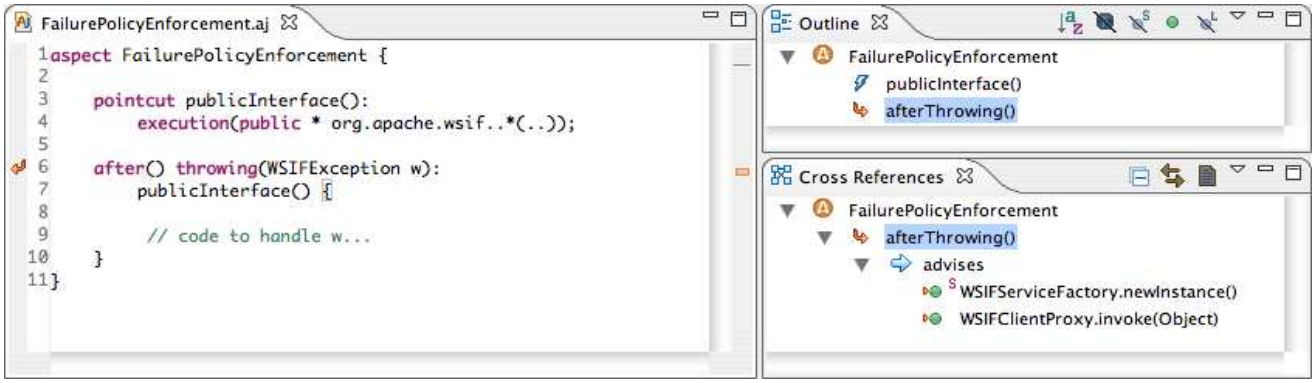


Figure 2. Advised-by annotation in the editor and crosscutting relationships in the Cross References view

When beginners start prototyping aspects their first challenge is to understand the places that the aspect affects. Advice affects the execution of join points. But whereas join points exist in the runtime call graph, the program elements that show up in IDE views correspond to static structure. As a result, the AspectJ structure views indicate all elements that could be effected by advice execution. We call these relationships between static program elements and join points the *join point shadow*. The join point shadow shows how advice affects program elements, and can be thought of as a projection of the dynamic execution of the program onto the static structure. To make it clear where the execution of advice will be decided by a runtime test (e.g. in the case of control flow advice) a question mark is added to the advice icon (Figure 4).

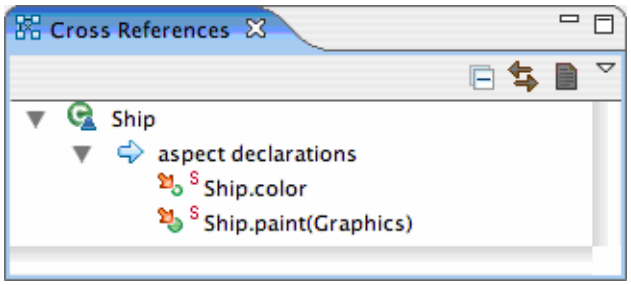


Figure 3. AspectJ declarations in the Cross References view

Some pointcuts crosscut very large portions of the system. Tree views are not effective for showing these because it is difficult to maintain context when scrolling through the hundreds of nodes that can populate the view. The Aspect Visualizer (Figure 5) addresses displaying the global effects of crosscutting by showing advice relationships in a zoomed-out SeeSoft source line-based view [5] inspired by the Aspect Browser [8]. The vertical bars represent source files, with height corresponding to file length. Each line affected by advice is colored according to the aspect-to-color mapping in the right-hand list. The Visualizer can display the crosscutting for an entire project, zoom into a single package, and navigate to the corresponding source code in the editor.

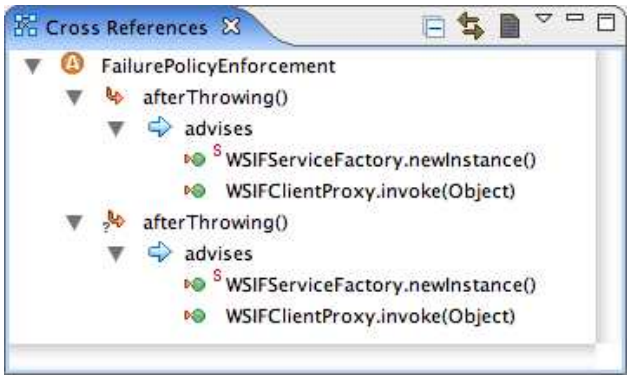


Figure 4. Advice with and without a runtime test

Java developers are accustomed to structured 'diff' views that indicate what parts of the object-oriented program have changed. As an aspect-oriented program changes, the changes in pointcuts can have wide-reaching impact on the system. For example, refactoring pointcuts may result in the associated advice affecting more join points than was intended, particularly if some of those join points were added by another team member. The Crosscutting Comparison view (Figure 6) addresses this by comparing the latest version of the program against a check point.

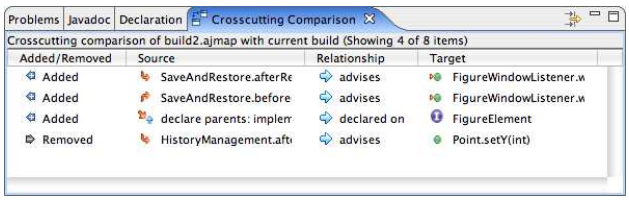


Figure 6. Crosscutting Comparison view

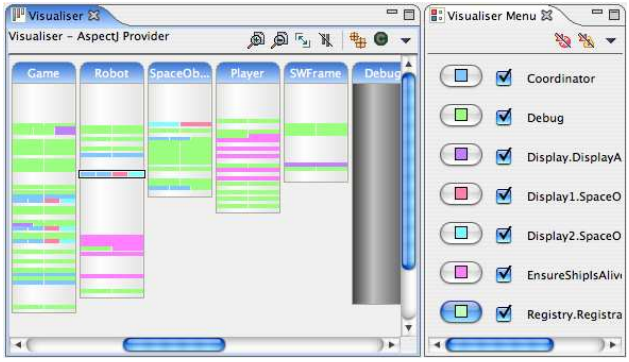


Figure 5. Aspect Visualizer

2.3 Build and editor support

Since we extended the Java language, the AspectJ language and compiler needed to be compatible with the Java platform. All legal Java programs are legal AspectJ programs, and the bytecodes produced by the "ajc" compiler can run on any Java 2 and later VM. Aspects can be declared in both ".java" and ".aj" source files. Pointcuts can be declared in classes as well as aspects, and inner aspects can be declared in classes. As a result, crosscutting modularity is as primary and visible to the AspectJ programmer as object-oriented modularity, and the AspectJ tools' role is to present a consistent view of both.

Integrating with the build process means that the AspectJ compiler provides semantic errors and warnings using the same problems list mechanism that IDEs use for Java errors. Since modern IDEs provide eager feedback on syntax errors, AspectJ editor support provides this for aspect members such as pointcuts and advice (Figure 7).

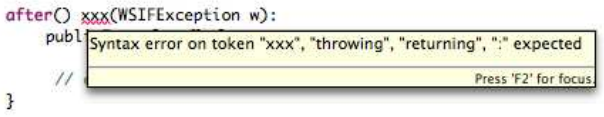


Figure 7. Errors as you type in the AspectJ editor

Additional integration between the compiler and the editor includes code formatting, the organizing of imports, and content assist (Figure 8).

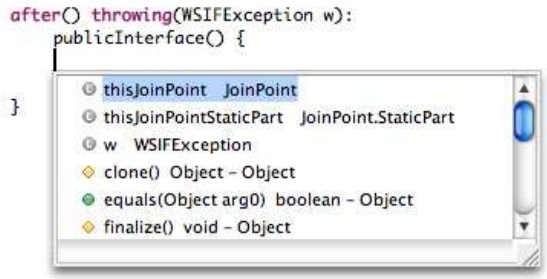


Figure 8. Content assist in the AspectJ editor

3. LESSONS LEARNED DISPLAYING CROSSCUTTING STRUCTURE

To expose crosscutting structure, OOP views needed to be extended to show the effects of aspects. Inheritance and encapsulation have clear representations in static structure views, but the core of the AspectJ structure model is dynamic join points. We struggled with ways to model and display an accurate aspect-oriented view of the program. This section reports the key lessons learned from trying to model and display crosscutting structure.

3.1 Crosscutting views made aspect-oriented programs easier to understand

User feedback from tutorials, workshops, and our mailing lists has indicated that the single most effective role that the tools have played is to expose the crosscutting relationships of the program. In contrast to OOP languages which have been successfully used with plain text editors, AspectJ depends heavily on tool support. A developer reading code written in an OOP language can understand the structure of the program by reading the text and following references. A developer reading code written in an aspect-oriented language can not infer the program from local examination of the code, since the crosscutting of aspects can affect the execution of that code. The increasing popularity and reliance on IDEs allowed us to leverage graphical views to make AOP program structure explicit (Figures 2-8).

The early releases of AspectJ IDE support made our users' reliance on the crosscutting structure views clear. At one point a bug prevented the views from displaying certain forms of call site advice, and resulted in confused reports asking if the language semantics had changed. During tutorials we often observed users learning the semantics of AspectJ by inspecting how the structure views show the effects of their first aspects. We have also observed users learning the syntax of pointcuts by inspecting the keyword highlighting and early error indication feature of the AspectJ editor (Figure 7).

3.2 New navigation techniques were required to expose crosscutting

Object-oriented views serve well for showing hierarchies and navigating references. But aspects are about non-hierarchical structure, and the presence of aspect-oriented structure in large systems can overload the tree views with relationships and links. In Eclipse the editor gutter annotations (Figure 2) suffer from a related problem. The gutter is narrow and only capable of displaying a single icon per source line. If additional annotations are present (e.g., breakpoint indicators) they occlude the advice annotation.

Since the Java IDEs showed program structure in tree views, we needed to extend these views to show crosscutting. But the advice affecting a method could not just appear as a regular child node of the method, since it relates through crosscutting and not by containment. Initially this resulted in our adding relationship nodes (e.g., "advised by") to tree views such as the Outline. To reduce the visual complexity and overload of those views when working on large systems, we moved the relationships out from the structure view and into the Cross References view (Figure 2). A keyboard shortcut also makes it possible to temporarily overlay this view on the editor, for example when a gutter annotation indicates the presence of advice.

Aspects are inherently good at expressing the global properties of a system, whereas object-oriented views tend to focus more on containment and information hiding. Changing a single pointcut can result in every public method of the system being advised. To show this global structure, the Aspect Visualizer (Figure 4) presents a cross-file or cross-package view of the crosscutting. The Crosscutting Comparison view (Figure 5) shows changes to the crosscutting structure between different versions of a project. This allows the programmer to see the crosscutting in the system not limited to a localized portion of the code, and shows the effects of refactoring a pointcut without needing to navigate to other pointcuts or advice.

We also had to provide facilities for helping the programmer maintain context when navigating crosscutting. When updating the body of an advice, the behavior of each of the join points it affects can be of interest. IDEs make it easy to lose context by presenting only the structure relevant to the current focus of the editor. To offset this we first introduced navigation history. Our tree links acted as hyperlinks and navigation could be stepped back and forward (most IDEs now support a similar history feature). This helped, but it did not solve the main problem of needing to see both aspect and affected join point at the same time. The Cross References view addresses this by maintaining a structure view focused on the crosscutting while allowing a second structure view, such as the Outline view in Eclipse, to be synchronized with the editor.

3.3 The tools taught both concepts of and misconceptions about the language

Early on in the development of the IDE support we decided to use Fluid Document technology [4] to present the effects of advice. Figure 9 shows how our prototype made the code of an advice appear at the applicable join point shadow with an animation that fluidly displaced the original code in order to prevent the programmer from losing context by being forced to navigate to the advice. This approach had the benefit of showing both the advice code, and the context in which it would execute. Unfortunately, this also gave the incorrect impression that AspectJ used preprocessor semantics and inserted code into a method, whereas advice does not execute in the same scope as

the method. As a result we did not proceed with this approach. This approach could have potential if combined with a mechanism that shows the separation of scope and the dynamic test that controls the execution of the advice. There is a related caveat with the Aspect Visualizer view (Figure 5) which encourages the developer to think in terms of source lines instead of dynamic points in the execution.

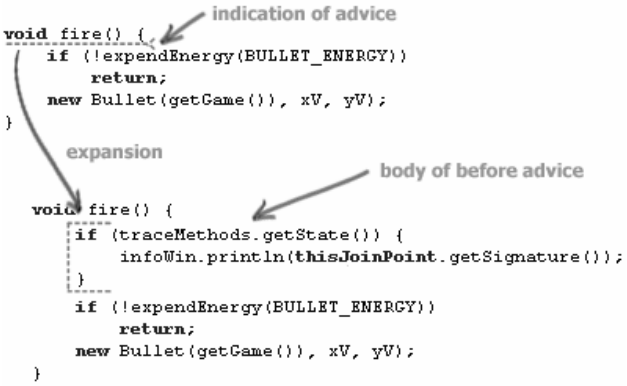


Figure 9. Fluid in-lining of a before advice body

Making it clear that AspectJ is based on a dynamic model has been the biggest challenge of displaying the structure of AspectJ's crosscutting mechanisms. When developers think in terms of static transformations of source code, they do not learn how to understand and work with aspects as a first-class structure of the system. We have repeatedly observed the preprocessor misconception preventing programmers from attaining an intuitive understanding of AspectJ's semantics. For example, it is awkward to think of join point parameter binding in terms of method parameters. As a result, a driving goal of crosscutting structure views was to discourage notions of inserting code.

3.4 Surfacing the dynamic properties of crosscutting was difficult

The current tools give the developer no assistance for determining what dynamic conditions will affect the potential execution of advice at a join point shadow. Multiple advice can apply to a single join point. Their execution is specified by ordering rules, or explicitly declared by the programmer. But the structure views do not surface the ordering semantics. There was no simple way to extend the object-oriented views to show this information, and it has remained an open problem[6].

Part of the problem stems from the fact that the IDEs' views show static structure. While much pointcut matching is static, which is why AspectJ is efficient, some matching has runtime tests (e.g., when a pointcut constrains the join points to only those within the control flow of a particular method execution). We use join point shadows (Section 2.2) to surface all of the places that advice might affect. For example, both method signatures and call sites are of interest for advice on calls. To make the programmer aware of dynamic tests we annotate the relationship with a question mark indicating the presence of the test (Figure 4). But the programmer is left to infer what runtime conditions need to be true for the advice to execute. The crosscutting structure views need to become more specific about providing context indicating what runtime tests and conditions will affect the execution of the advice (e.g., by showing the call graph for the control flow constraining a pointcut). Exposing the dynamic properties of crosscutting will involve extending our structure model, which like the IDE's structure models has focused on representing structure that can be easily mapped to source code.

3.5 A crosscutting structure model was key, but over-generalizing it was a mistake

From the beginning of the project our use of agile methods helped the tools evolve along with the changing language implementation and IDE platforms. The tools framework grew out of a single IDE implementation, to one generic enough for two Swing[7]-based IDEs, and finally to a GUI-independent framework when we needed to support Eclipse's SWT toolkit.

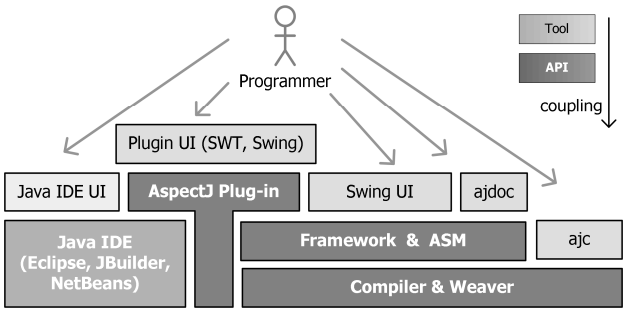


Figure 10. AspectJ tools framework and clients.

The AspectJ Structure Model (ASM) is at the core of the tools framework on which the IDE plug-ins are built. ASM clients expose the model in task-specific views. Unlike the compiler's abstract syntax tree (AST), the model is kept in memory since the crosscutting structure of the entire system must be presented to the user without invoking a search. If the user switches build configurations it is saved to disk. Figure 10 shows how our tools can extend the ASM directly, as ajdoc does, or extend the framework to provide views specific to the look-and-feel of the host IDE. The Framework and ASM have also been extended by someone outside of the core AspectJ team in order to provide support for the JDeveloper IDE[8].

The ASM represents the crosscutting, inheritance, and referential structure of AspectJ programs (Figure 11). Since structure views represent static program structure, the structure model is a projection of AspectJ's dynamic properties onto the static structure of the system—i.e. from the join points to the join point shadows. The example in Figure 11 depicts pointcut

[6] For potential solutions see http://eclipse.org/ajdt/ui.html

[7] http://java.sun.com/products/jfc

[8] https://jdeveloperaop.dev.java.net, created in 2004

and advice relationships. Note that the pointcut does not point to affected members, since a pointcut alone does not have a behavioral effect on the program.

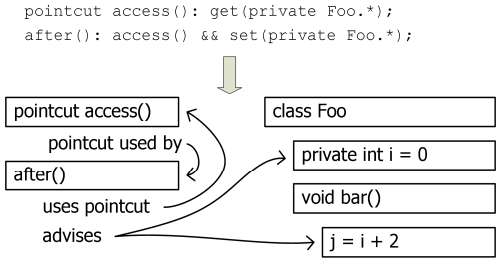


Figure 11. Example of ASM relations

The ASM started out as a string-based map of correspondences between declarations. To incorporate structural relationships for inter-type declarations and other kinds of join point shadows we built a generic data structure composed of program elements and associations relating them. This graph was used for constructing containment, inheritance, referential and crosscutting views. It did not contain a dominant decomposition (the containment hierarchy in most IDEs) and did not require searches to build structure views. It also captured additional relationships, such as the "@see" links from Javadoc. It was intended to be extensible to UML[9]-specific associations or relationships between program elements and structure declared in external resources (e.g., XML[10] files in J2EE[11]). We also started down the road of making the ASM general enough to support a wide range of AOP tools and languages. However, over-generalizing in this way turned out to be a mistake.

The latest ASM implementation is, once again, a string-handle-based mapping between elements in a containment hierarchy. The more flexible and extensible generic data structure attempted to solve interesting problems, but was not practical for solving the core problem of supporting high-quality IDE integration. The lack of a dominant hierarchy resulted in multiple graphs representing the entire program structure and causing an excessive memory footprint when used with systems larger than ten thousand classes. In addition, to achieve a deep integration with the Eclipse IDE the update of the structure model needed to happen eagerly as the user edited. In order to make the ASM support incremental update and improve performance we concretized it and returned to using string handles as references to program elements. The relationships still persist in memory, but as a much leaner and easier to incrementally update map between handles corresponding to program elements.

[9] http://www.uml.org

[10] http://www.w3.org/xml

[11] http://java.sun.com/j2ee/

4. LESSONS LEARNED EXTENDING OBJECT-ORIENTED IDEs

Modern IDEs provide the programmer with a lot of support for editing and navigating object-oriented programs. We needed to extend the existing functionality in a consistent and elegant way to support AspectJ without getting in the way of the Java tooling. In building this support we iterated through three levels IDE integration, and each of our tools evolved along the path defined by Table 1.

Table 1. Levels of IDE integration

Integration	Goal
1. Invocation & resources	Invoke the AspectJ compiler on project resources and display compiler messages.
2. Editor & views	Provide a custom editor for aspects and new structure views.
3. Structure model	Integrate with the IDE's structure model, editor, parser, and views, and add new crosscutting-centric views.

The *invocation & resources* integration was our first improvement over the command line compiler and text editor interface. This involved mapping the IDE's definition of projects and paths to command line parameters understood by our compiler. This level of integration is still used in the form of Ant support by those who do not have IDE support for AspectJ (e.g. IntelliJ users).

The *editor & views* integration improves on the previous level by providing an editor that understands aspects and can support features like keyword highlighting, and views that show crosscutting structure. The JBuilder, NetBeans, and early AJDT plug-ins offered this level of integration. For example, the Outline view in these looked very similar to the IDE's Outline view, but was a replacement that showed AspectJ declarations and crosscutting relationships. But this approach is fundamentally limited because the advanced IDE features that rely on the IDE's parser, compiler, and structure model are not supported, and views that mimic the IDE's views are never up-to-par with the user's expectations.

The *structure model* integration extends the core model of the IDE to understand AOP semantics. AJDT provides this, and exposes it in features such as eager parsing and content assist. But deep integration is challenging due to the Java-language specific bindings and assumptions made by the IDE's structure model. As a result, as of the writing of this paper AJDT does not yet integrate deeply enough to offer refactoring support, which is now a commonly expected feature in Java IDEs.

While working through these levels of integration we have struggled with extensibility limitations that result from AOP breaking the core assumptions made by object-oriented IDEs. OOP is about encapsulation and hierarchies, whereas AOP is about structure that crosscuts encapsulation and hierarchies. As a result the underlying data structures of AOP and OOP tools are fundamentally different. In this section we report on the strategies that we developed for providing a clean integration of AOP into OOP IDEs.

4.1 Overriding the file extension broke the IDEs' extensibility model

AspectJ is intended to be a seamless integration of AOP and Java. The file extension for AspectJ sources seems like a seemingly small detail, but turns out to have significant implication on the degree to which the integration is seamless. One of the core goals of the AspectJ language is to make crosscutting mechanisms available as a part of the base language. This goal is in contrast to the reflective and XML/annotation-based approaches (e.g., AspectWerkz[12]) in which crosscutting is declared outside of the main language. AspectJ's approach benefits the programmer by providing consistent support for both objects and aspects. However, current IDEs only allow for language extensibility outside of the core program text (e.g., in comments, metadata tags, strings, and new file extensions). In any file named ".java" the tools expect a language that conforms to the Java language specification.

The easiest way to address this is to only allow AspectJ code in separate resources (e.g., ".aj" files with the current syntax or ".xaj" with XML syntax). But this would have only side-stepped the problem. For example, consider an inter-type declaration made in a separate resource. The tool support could try to make the resulting structure clear, but there would still be a confusing and awkward disconnect between the external code for the inter-type declarations and the pure Java declarations. A reference to an inter-type declaration in a ".java" file would be a compiler error. The AspectJ language and tool support makes crosscutting a primary part of the system's architecture and allows aspect declarations in ".java" files.

AspectJ 5 offers a compromise called the @AspectJ style (vs. the code style of ".java") first introduced by AspectWerkz. The @AspectJ style allows declaration of aspects and aspect-members using pure Java syntax by exploiting Java 5 style annotations. This is less disruptive to tools expecting pure Java syntax inside ".java" files, but does not alleviate the major requirement to show crosscutting structure. The @AspectJ style was released recently, and we are still gathering feedback on this approach. But a key enabler is the fact that AJDT can present crosscutting consistently in both styles, and even allow toggling between one style and the other by rewriting the aspect.

4.2 No IDE was inherently extensible to AOP

This may seem at odds with the fact that we built IDE plug-ins. But the AspectJ plug-ins do not directly extend the OO structure model of the IDEs. They either replace it or layer onto it. When contrasted with the efforts behind commercial Java tooling, the resources available to the AspectJ project have made this catch up game a slippery slope. Take for example the Eclipse IDE, which offers no facilities for extending its Java model with new semantics. The structure model is the foundation of features such as eager parsing, code assist, and refactoring. Both the core Java tool support and 3rd party extensions contain explicit bindings to the Java language, and as such do not inherently support language extensions. Taking the external-language approach is less intrusive to the other Java tooling (Section 4.1). But this only delays the problem by pushing the aspect-oriented structure out of the way. IDEs' structure models needs to be made semantically extensible to other program structures in order to cleanly integrate crosscutting. So far we have only to layer on this sort of extensibility to the Eclipse IDE, which is open source and has a rich model of Java Structure. As a result, the core AspectJ team's integration efforts are focused on Eclipse.

4.3 Semantic extensibility requires openness and structure-aware features

Our first IDE plug-in after Emacs extended the Microsoft Visual J++ 6 IDE[13]. Its extensibility APIs are similar to those currently available in VisualStudio.NET. However, the IDE was closed-source and not self-hosted on the APIs (i.e. the internal implementation was not based on them). It was not possible to extend the IDE beyond the limited use cases that the IDE's developers had planned for. The release of the pure-Java JBuilder 3.5 was promising. We released the first IDE plug-in on JBuilder's much broader open APIs. The breadth and stability of these APIs helped bring AspectJ into the hands of real developers [14], but the APIs were not designed for incorporating new modularity ideas and language extensions. JBuilder is not open source, and was missing critical extension points. For example, to make the file structure view work we had to walk the Swing component tree, and overwrite the Java structure specific view.

The first NetBeans release was encouraging since the IDE was open source. Open sources proved invaluable for figuring out how to extend the IDEs in a way that was not originally planned. However, due to an overly general architecture and not enough focus on extensible Java tooling it was more difficult to build the NetBeans plug-in than to build the JBuilder plug-in. The Eclipse 2 release combined openness, broad APIs, a much deeper self-hosting on those APIs, a more complete plug-in component model, and more features for viewing and manipulating OOP structure. Eclipse was also the only IDE to provide an open source compiler, AST, and structure model, which offered opportunity for a deep integration. However, the advanced tool features of Eclipse are a double-edged sword for language extensions. They set the feature bar very high. For example, Eclipse users new to AspectJ often expect refactoring to work for AspectJ as seamlessly as it works for Java.

4.4 The cost of supporting multiple IDEs was justified by a broad outreach

We released support for 7 versions of JBuilder, 5 versions of NetBeans, and 4 versions of Eclipse. Supporting multiple IDEs was costly. But it helped establish AspectJ as standard for AOP on Java rather than as an extension to a single IDE, and helped ensure that the command line compiler remained de-coupled from any IDE. Also, we could not initially choose one IDE up-front because we did not know what our users' platform of

[12] http://aspectwerkz.codehaus.org

[13] http://msdn.microsoft.com/vjsharp

choice would be. More recently we have prioritized deep integration with a single IDE, but have maintained the APIs that make extensibility with other IDEs possible (Section 3.5).

The porting of plug-ins to newly released APIs of a given IDE has been more straightforward than we expected. Our main problem has been maintaining a single release that is backwards compatible with older releases. Users want the latest bug fixes even if they are stuck on an older version of the IDE, while others expected immediate support for new IDE releases. Supporting both the latest and older versions imposed a drag on evolution, was time consuming, and involved marginal solutions such as checking the IDE version at runtime and invoking the appropriate API call reflectively. The only complete solution is to maintain multiple release streams, one for each IDE release, and this is the approach we have resorted to now.

One good domain for AOP is enterprise applications [3]. To enable AspectJ use for these developers we needed to support the enterprise versions of the IDEs (JBuilder® Enterprise Edition, Sun Java Studio Enterprise built on the NetBeans platform, and IBM® Rational® Application Developer for WebSphere® Software built on the Eclipse platform). In theory, this should not have required extra work since all plug-ins making correct use of the extensibility model should interoperate with the extended enterprise IDEs. However, testing and updates specific to the enterprise editions were necessary to ensure compatibility. This additional work was necessary, in part because enterprise project configurations involve more kinds of resources, and because the enterprise editions can depend on core platform features that are not used by the standard edition. Unlike the early adopters who always downloaded the latest free standard version of their IDE, the enterprise developers are often stuck on a version until their organization decides to upgrade. Another problem was that we did not use the enterprise versions of the tools internally. Close dialog with enterprise versions users helped us understand the expectations they had of the integration.

4.5 Integrating with existing UIs was more important than creating new ones

Numerous IDE views can be affected by the presence of aspects, for example:

- Document outline: additional members
- Content assist: additional members and special join point variables
- Inheritance tree: additional super types can be declared by aspects
- Debugger thread tree: additional stack frames appear for generated methods

Our bug report database indicates that the most important part of clean integration is not getting in the way of existing Java tool support, but showing the relevant crosscutting information when needed. New users curious to try AspectJ are not willing to continue using it if they have to sacrifice their existing Java support. To further improve integration quality we set a policy of zero-configuration after install (e.g. by automatically configuring preference settings). In addition, we had to provide IDE-specific UIs for toggling whether or not AspectJ was enabled for a particular project. Although effort spent on improving integration was at the expense of adding features it was critical for getting feedback from use on real systems. For example, the first release of JBuilder support with zero-configuration resulted in many more bug reports than we had ever received for the tool. This made us realize that users were not bothering to submit bugs if they did not get far when first trying the tool.

We started by making AspectJ features as visible as possible (e.g., there was an AspectJ menu that provided build commands). As our plug-ins improved, access to AspectJ features was integrated with the corresponding Java features. In general, extending existing views worked better than adding new ones. We were able to populate the standard Outline view with the structure of aspects, instead of using a custom version of the Outline view which did not look or behave identically to the standard version provided by the IDE. The new structure is placed inline with Eclipse's editing and navigation features, and indicates when the user might want to pop up a view or menu indicating the crosscutting structure. We only provide additional views (Figures 4-6, Cross References, Crosscutting Comparison and Visualiser) to support crosscutting-centric navigation of inspection of the system.

4.6 Integrated debuggers were more extensible than expected

In 2001 we released a JPDA[14]-based debugger with a command-line and GUI interface. However, we were not able to get the quality and features of the debugger up to the expectation set by debuggers in existing IDEs, and discontinued the standalone debugger after 1.0. Extending existing debuggers turned out to be easier than extending the IDE's core structure model because of the extra extensibility that results from requirements on debuggers to support breakpoints in non-Java languages. For example, to support Java Servlets, a debugger must map bytecodes to corresponding Java source embedded in ".jsp" files (JSR-45[15]).

We were able to support the use of standard Java debuggers on AspectJ by encoding the source line mappings in the bytecodes. However, Java debuggers expose the implementation of the AspectJ language instead of the language semantics. For example, extra frames corresponding to advice call-outs show on the stack. Nevertheless, this allowed us to consider the additional functionality as an additional user interface layer over the existing debugger functionality. User feedback on debugger support has indicated the need to preserve both views. The standard Java debugger views turns out to be useful for the cases where seeing exactly what executes is more important than seeing clean AspectJ language abstractions, for example in resource-constrained environments.

[14] http://java.sun.com/products/jpda

[15] http://www.jcp.org/en/jsr/detail?id=45

5. SUMMARY

AspectJ provides tool support that exposes the aspect-oriented structure of programs by extending object-oriented IDEs. In this experience paper we reported the way in which we surfaced crosscutting structure by extending the object-oriented IDEs. From our user community we learned about the need for seamless integration with the IDE, the importance of providing mechanisms for viewing the global effects of aspects and the need for maintaining context when navigating crosscutting structure. Whereas we succeeded at showing the static mapping of join points to structure views, we have not yet managed to fully expose the dynamic properties of crosscutting. We also learned to choose mechanisms for displaying crosscutting carefully, since these have the ability to both teach the language and to encourage misconceptions about it. The most difficult problems we encountered resulted from the lack of extensibility beyond OOP that is an inherent limitation of the tool platforms. But we are continuing to improve the depth and integration of the AspectJ tool support, and in the process hope to provide additional experience on making object-oriented tool platforms more extensible.

6. ACKNOWLEDGEMENTS

Thanks to Gregor Kiczales and Gail Murphy in helping select and distill the lessons learned, the AspectJ and AJDT teams who continue to improve the tool support, and currently include Adrian Colyer, Jonas Bonér, Andrew Clement, George Harley, Helen Hawkins, Wes Isberg, Sian January, Mik Kersten, Alexandre Vasseur, Julie Waterhouse Park, and Matthew Webster. Additional thanks go to Erik Hilsdale and Jim Hugunin, former members of the AspectJ team, who played a key role in helping design the tool support, and to our user community whose feedback continues to teach us about AOP.

7. REFERENCES

1. Beck, Kent: Test-Driven Development By Example, Addison-Wesley, 2003.
2. Colyer, A., Clement, A., Bodkin R., Hugunin, J.: Practitioners report: Using AspectJ for component integration in middleware. In: Proceedings of the Conference on Aspect-Oriented Software Development (AOSD). Lancaster, UK (2004)
3. Colyer, A., Clement, A.: Large scale AOSD for middleware. In: Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). ACM, Anaheim, California (2003)
4. Chang, B.W., Mackinlay, J.D., Zellweger, P.T. and Igarashi, T.: A Negotiation Architecture for Fluid Documents. In: Proceedings of the ACM Symposium on User Interface Software and Technology, pp. 123-132 (1998)
5. Eick, S.G., Steffen, J.L., Sumner, E.E.: Seesoft - A Tool For Visualizing Line Oriented Software Statistics. In IEEE Trans. on Software Engineering, Vol. 18, N. 11 (1992)
6. Friendly, L.: Design of Javadoc. In: The Design of Distributed Hyperlinked Programming Documentation (IWHD). Springer-Verlag, Montpellier, France (1995)
7. Gosling, J., Joy, B., and Steele, G., Bracha, g.: The Java Language Specification. Second Edition. Addison-Wesley, Reading, Maschusetts (2000)
8. Griswold, W.G., Kato, Y. and Yuan, J.J.: Aspect browser: Tool support for managing dispersed aspects. In First Workshop on Multi-Dimensional Separation of Concerns in Object-oriented Systems, OOPSLA (1999)
9. Kersten, M., Murphy, G.: Atlas: A Case Study in Building a Web-Based Learning Environment Using Aspect-Oriented Programming. In: Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). ACM, Denver, Colorado (1999)
10. Kiczales, G., et al.: An Overview of AspectJ. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP). Springer-Verlag, Finland (2001)
11. Kiczales, G., et al.: Aspect-Oriented Programming. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP). Springer-Verlag, Finland (1997)
12. Moon, D.: Object-oriented programming with Flavors. In *Conference on ObjectOriented Programming Systems Languages and Applications*, pp.1-8 (1986)
13. Masuhara, H. and Kiczales, G.: Modeling Crosscutting in Aspect-Oriented Mechanisms. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP). Springer-Verlag, Spain (2002)
14. Price, R.: Real-world AOP Tool Simplifies OO Development. Java Report, September Issue (2001)

Gathering Feedback on User Behaviour using AspectJ

Ron Bodkin
Founder
New Aspects of Software
rbodkin AT new aspects.com
(650) 941-8344

Jason Furlong
Consultant
New Aspects of Software
jfurlong AT new aspects.com

Abstract

This report describes our experiences applying aspects to provide feedback on user behavior, system errors, and to provide robust a solution for a widely deployed diagnostic technology for DaimlerChrysler. We describe how the capability is being extended to support recording and playing back macros. We detail a list of challenges and lessons learned as well as report on the tools that we used to build the feedback feature.

These aspects have been incorporated in production releases and are used to improve the application and underlying business process performance in a deployment to thousands of locations. We achieved our primary objective of being able to seamlessly integrate the User Monitoring concern with minimal coupling to the core application.

Overview

New Aspects of Software has been consulting and training for the DaimlerChrysler Next Generation Scan Tool (NGST) project, providing expertise in aspect-oriented software development in particular and in effective application architectures and agile development more generally. This experience report describes our experiences using AspectJ with an existing Java GUI application that is distributed to a large number of dealerships and other service centers.

The NGST project delivers a family of hardware and related software applications for use in diagnosing vehicles. The first generation product, the StarSCAN, is an embedded device running Linux and a Java 1.1 VM using AWT. The second generation product, the StarMOBILE [StarMOBILE], is a device that can run Java applications in standalone mode, or in a pass through mode in which a “Desktop Client” Java 1.4 application controls the behavior. The NGST project wanted to implement an “in flight data recorder” feature that would track the user’s interaction with the Desktop Client application to better understand how users used the system. Once gathered from all deployed versions, this data would be used to optimize the user interface. Capturing this data is especially timely as it will be used in a new effort to update the NGST user interface from one suited for use on embedded devices to one that makes use of a full featured desktop computer.

This report describes our experiences in developing and deploying this feature, which has recently been released globally to thousands of locations. This project represents an important step in the adoption of aspects on this project. Subsequent to our initial effort we have reused the feedback pointcuts to capture and play back macros. Future plans include developing aspects to provide rigorous integration tests of components and to support product line variations for different software configurations.

Approach

The project followed an iterative development process. We started with a simple summary of requirements derived from discussions with the project stakeholders. Two developers on the project, Sam Konyn and Guangjing Zhou, had previously written aspects to allow recording system events and output for subsequent playback in regression testing. Using their experience, we devised a

preliminary set of pointcuts for monitoring the application. These pointcuts formed the core of a set of tracing aspects that were incrementally refined to determine which join points would best capture GUI events.

Initially, we had expected to track user events (button pushes) as well as navigation among UI views. However, for this application, it became clear early on that the most interesting information was defined by the views since there was typically only a single way of navigating from one view to another. Moreover, the UI events in the application, like the UI approach in general, was somewhat complex and had evolved over time, with newer code following different patterns. Once we decided to focus on the change in views that were presented to the user, we were able to significantly reduce the amount of data that we were writing to the log. This provided us with a robust set of pointcuts to capture the transition of main screens but didn't capture several special cases. Foremost among the special cases were wizards that were a combination of AWT and Swing-based panels.

To arrive at robust pointcut definitions, we needed to address a number of issues. The system in question often relies on events to invoke operations, yet there are significant variations in how events are handled and distributed. We elected to depend on the executions of certain non-public but descriptive navigation methods rather than the nuances of event. Most of the pointcuts depended on base class methods, rather than enumeration or use of name patterns. In a few of the more complicated cases, we needed to correlate a series of join points, tracking state with inter-type declarations. We consciously avoided any use of privileged aspects and took care to minimize dependencies on parts of the system that were deemed likely to change. In performing some *judicious* refactoring to some of the existing UI code we exposed relevant information and made some protected methods public. This resulted in an improved integration of information with user monitoring. Additional pointcuts were needed for the tracking of metadata (both device and vehicle) that exposed the connection state of the application.

As an additional feature of the feedback concern, we included pointcuts to capture a summary of exceptions encountered during program execution. Advice attached to the error monitoring pointcuts tallies a count of exceptions based on certain common characteristics (message, subclass of Throwable, and the join point that first handled them). Upon system shutdown, the monitor aspect writes all the error statistics to the log.

Once the implementation was working well, we refactored the experimental aspects into a production-ready design that divided up responsibilities by subsystem. We created an abstract base monitoring aspect that was extended by several concrete aspects; each one concerned with monitoring specific types of user interface or other events. The system also uses an aspect to ensure that session information (metadata) is included with each log output entry.

Error Isolation

We created an Aspect whose job was solely to contain any errors generated within the monitoring code so that they wouldn't flow into the basic application. This was inspired by Ron's previous work with Glassbox [Glassbox], which demonstrated the importance of isolating errors in monitoring code and showed the feasibility of doing so with aspects.

Configuration and Runtime Control

As the NGST application uses an Inversion of Control (IoC) container[1] strategy for

[1] Specifically, it uses the Interface21 framework, which is a precursor to the Spring Framework. Interface21 is still being used because of its

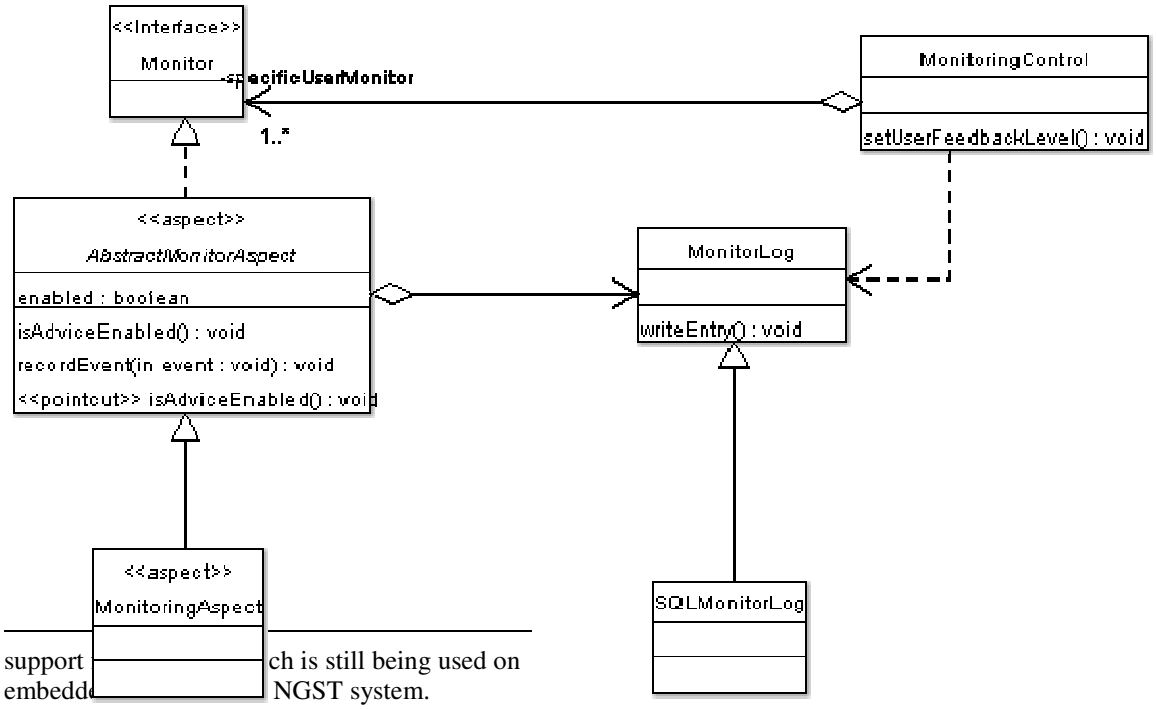
configuration, we elected to utilize it to configure the feedback feature. All the aspects associated with the feedback concern are configured through a class which acts as a configuration façade that delegates the setting of values. Initially, this design was chosen to work-around limitations in the IoC container's configuration ability (in marked contrast to the full Spring 1.2 framework). When capturing the system's state we had a choice between using the existing configuration mechanisms or writing pointcuts to find the unique instances of objects being created. Ultimately, we preferred the use of an IoC configuration, to minimize coupling and to keep the mechanism consistent with the rest of the application.

A customer-defined requirement of the system was the ability to enable and disable aspects at runtime. There are two means of configuring this ability: a user can explicitly change the feedback level in the application, and when updated, the system as a whole can be configured to change the level allowed.

Subsequently, the configuration façade proved useful to provide a single point of control for monitoring policy changes through the enabling and of disabling aspects. The abstract base monitoring aspect defines an interface for enabling and disabling advice, and we reuse an idiom for runtime control of aspects that is described in [Glassbox2]. The monitoring control bean can iterate over each aspect, configuring the level of feedback required.

The monitoring control bean also is responsible for managing the life cycle of the shared event log (including registering and unregistering a shutdown hook). It also registers as an event listener for some system events in the application. It further *publishes* an event for the UI screen to listen to. We discuss the use of the Observer pattern below.

To support macros, we are refactoring this approach to split responsibilities so we can support both logging events and updating display state. We have defined thin monitoring aspects that track various view events, system events, and tracking error. These expose a *pointcut interface*. I.e., logging and updating display state aspects can advise these pointcuts. However, the context data we want to expose from the join



support ch is still being used on embedd NGST system.

Figure 1. Configuration of Monitoring Aspects

points can not be bound using AspectJ pointcuts. Instead, the monitoring aspects track the system's state and create an *arranged join point* that passes the relevant context to a no-op method. This allows the two different types of aspects to simply write advice before a view display event is seen:

```
public interface Monitor {
   void recordBeforeViewDisplay(ViewDisplayEvent event);

   pointcut beforeViewDisplay(ViewDisplayEvent event) :
      execution(* Monitor.recordBeforeViewDisplay(..)) &&
      args(event);
…
}
```

Testing

We performed unit tests of the core responsibilities of the monitoring aspects. However, due to the complex, inconsistent and undocumented UI structure of the system, integration testing was the most essential requirement. The monitoring aspects are broken into a number of cases, mirroring the various different implementation approaches used.

Nicholas Lesiecki has coined some good terms to describe things that might possibly break when writing aspects. In his terms, we had to test carefully to find *Unheeded Advice* and *Unwanted Advice* (including the special case of *Duplicate Advice*) and in some cases *Bad Binding* (from drafts of [LesieckiTest]). While writing unit tests to verify correct behavior against a model of the general pattern of system behavior is useful, it is much less effective than systematic integration tests that ensure the model is faithful. We have written integration tests that verify both behavior and interfaces for the feedback subsystem (e.g., ensuring that changes to the runtime configuration enable and disable advice and produce proper logging). We integrated the JMock framework for proxy-based mock objects and also have used virtual mock objects for integration testing (e.g., to replace VM shutdown hooks with actions at the end of a test case). [Ajmock]

However, the project is currently focusing on the use of FitNesse [FitNesse] to allow regression testing of system functionality. We have designed an approach to integrating feedback testing with any FIT tests: we will ensure that the fixtures used to simulate user input are also monitored correctly by the feedback system. Then we will make a crosscutting assertion that can affect any FIT test. We can check the monitored output for any or all system acceptance tests & report back if there is expected output and the actual output diverges. The comparison will mostly involve comparing for file equality, although a few fields (time stamps and session id's) will need to be extracted out as variables. As existing FIT tests are completed, we will be in a position to add integrated feedback testing to whatever degree we need to mitigate risks. As a result, the project conducted extensive manual system tests, which will provide input to automated testing in future.

Tools Integration

The project team had standardized on the IntelliJ IDEA IDE for development and uses Apache Ant for production builds. The first approach we used to develop aspects in this environment was to extend the Ant build scripts to allow certain modules to use the AspectJ compiler (ajc) instead of a Java compiler for compiling production code and tests, and to allow tests to weave into production code to support virtual mocks. The ant build script built separate, unwoven jars for each module, with the intention being to weave subsequently.

The next step we took was to set up load-time weaving so that aspects could be applied to the code that IntelliJ compiled, using the standard IntelliJ launcher. To accommodate this, we developed a custom load-time weaving implementation that works with AspectJ 1.2.1 or the AspectJ 5 milestone releases. This implementation uses a non-delegating URL Weaving ClassLoader that weaves all the classes it

loads by default, to allow easy integration with the IntelliJ IDEA IDE. The AspectJ URL Weaving ClassLoader uses a different classpath variable to specify the classes to be loaded, which wouldn't allow us to use the standard IDEA mechanisms to assemble classpaths when launching.

However, we really missed having autocompletion, integrated compiler output, and the ability to navigate project structure inside the IDE. So we set up a limited scale parallel development environment in Eclipse using the AJDT plug-in. Even given some bugs and limitations that are discussed below, the environment was more productive. Initially, we developed aspects by depending on ant-built jars that would be included in the inpath of the feedback project in Eclipse.

Subsequently, we presented the benefits and costs of switching from IDEA to Eclipse to the team. The major benefits from this would be better support for developing aspects in particular and the much greater level of industry support for Eclipse in general. The team decided to try using Eclipse again, and Guangjing Zhou set up an Eclipse workspace for development. This has been a much more productive environment for aspect development, since it allows integrated development with the Perforce version control system the project uses and allows working on all the code in one environment. However, the environment uses linked source folders for all projects, which has caused a number of problems which we expect to be resolved by using direct references to file system folders.

We have also sponsored Mik Kersten who has developed better support for incremental ant compilation. The idea is to allow an IDE like IDEA to kick off incremental ant builds and to show crosscutting structure with visualization views, to better allow developers using IDEA to see the affect of aspects in their environment. We think this tool will be helpful as developers want to see how aspects affect Java code, but that developers writing Java code will still want to have the productivity of integrated IDE support with easy navigation to AspectJ types and code completion when writing aspects.

There have been a few minor issues in tools integration to date. When the AspectJ ant task produces compilation errors, the CruiseControl automated build system doesn't handle them like Java compiler errors and display them in the summary of errors encountered. Instead, we have to open up a detailed log viewer to find the compiler output. We have spent some time investigating this issue, integrating an update to ant support after AspectJ 1.5.0 and finding problems in reporting ant tasks from that. A bigger potential problem is the use of the Clover test coverage tool, which works only on Java source code and has to be excluded from tracking coverage in AspectJ code. While we have used bytecode-based coverage tools (notably EMMA) on past projects successfully, this project adopted a different source code coverage tool, and it is typically a problem to switch technologies.

Performance

While we did not measure runtime performance precisely, there was no noticeable delay in the performance of the monitored code during testing. There was, however, a noticeable delay at start up when we tested the use of load-time weaving to apply the monitoring aspects (e.g., taking 10 seconds instead of two seconds on a minimum recommended workstation for the application).

It is important to minimize the bandwidth required to update binaries for each release, to allow quick updates even over shared connections at 100 kbps. Using build-time weaving would have resulted in a one time complete replacement of the entire application. Moreover, changes to monitoring aspects would have likely required similarly large updates to many distributed components for future releases.

Requiring such extensive updates was highly undesirable.

As such, we created a custom utility that performs command-line weaving for AspectJ requiring only AspectJ's 1.5 megabyte aspectjweaver.jar (which designed for load-time). This utility consists of a stripped out and simplified weaver, which performs the same binary weaving functions as the AspectJ ajc compiler, but avoids the need to distribute a 4.5+ megabyte compiler jar with source code compilation capabilities. This utility has been contributed to the AspectJ project and is slated to be integrated into AspectJ. This command-line weaver was integrated into the existing installation scripts, to weave each jar as it is unpacked. This approach allows correct functioning when the application is partially updated.

Assessment

Benefits

The primary objective was achieved as we were able to seamlessly integrate the User Monitoring concern with minimal coupling to the core application. The minor changes that we implemented in the core application actually contributed to making it more modular. For example, the feedback system invokes a vehicle scan report that captures the current state of the vehicle being diagnosed. In the original application this feature was tied into the GUI process, but the new requirement of being called from a headless required one of the team members to separate out the Report Generation from the GUI code.

DaimlerChrysler will now be able to gather metrics about application use to streamline future versions of the user interface. In many respects, this data collection allows analysis of user behavior like Web log analysis packages do, although there is the potential to capture more precise information in a rich client environment. In particular, we expect to analyze the most frequently used views, types of problems and data that are relevant, and common navigation paths. These metrics will assist in a reorganization of the presented views, and the order in which they are presented which should facilitate training and better creation of content to improve diagnostic effectiveness. It will also indicate areas in which shortcuts could be best introduced in the application, views that should not be buried deep in a hierarchical view set as well as profiles of different users. Monitoring the use of the embedded tools that the end-users use allows the client to determine which diagnostic procedures are effective and which ones are seldom used. The monitoring system will also capture more accurate information about errors experienced in the field, including errors that are intermittent that may not be reported. All the metrics are uploaded periodically to a Web server and then loaded into a database server, from which standard reports will be generated along with some ad hoc analyses.

The feedback subsystem consists of 1200 lines of production AspectJ code distributed amongst 10 aspects plus an additional 2400 lines of supporting classes. The feedback system affects about 8500 join points in the rest of the system. The NGST family of software contains over 400,000 lines of Java code.

This represents a major improvement over what would have otherwise been an extremely scattered and tangled implementation. If a traditional implementation had been performed in this case, then code would have had to been inserted into the core application in all these places, and it would have been difficult to implement a consistent and correct strategy. Moreover, it would have been difficult to follow the incremental refinement strategy to feedback that we plan to use. This would have further required much greater preplanning, more conflict resolution, and more coordination and communication with the other programmers on the team. Because we were not adding code to core

components of the system we nearly eliminated all conflicts and did not introduce additional errors into the system. Any additional errors that we did introduce with the aspects were contained within the error monitoring aspect.

An important benefit of the AOP solution is that it can be unplugged for certain configurations, and updates to feedback (e.g., to track user events) can be made modularly. The localization of the user monitor feature within a several aspects as meant that configuration changes of the monitoring policy was easy to manage both during runtime and at startup. The entire feedback system was contained within its own package hierarchy in a separate module. The only code that was not contained within this separate module was the user interface to allow user configuration of the feedback system.

Because we separated the user monitoring concern from the rest of the application we reduced the need to add monitoring to all the unit tests for the target classes. As the project is following an agile methodology, unit tests form a very important part of the software development process. Avoiding the need to rewrite all the unit tests represents a significant cost reduction for the installation of a new feature. As noted above, we were able to unit test the feedback feature in isolation and decide how much integration testing to perform in a principled fashion.

The rest of the team continued to develop new features to include in the next release and wanted to include the feedback feature with these. By using AOP we were able to work fairly independently, with few conflicts and no need to enforce a policy for instrumenting all new application code. AspectJ significantly reduced the need for coordination with the other programmers. The areas in which we did need to coordinate were in understanding the UI patterns of the application and in integration of specific capabilities like file upload, reporting, and specific data collection. This is typical of integrations between cohesive modules, again indicating that the feedback capability was well modularized.

Overall, the AspectJ tools worked well. The command line AspectJ compiler (ajc) and its integration within Apache Ant worked very well. While we needed to do some work to integrate AspectJ technologies into the build and installation process was fairly straightforward. The debugging support within Eclipse was also of great benefit and the AJDT plug in for Eclipse was very useful, even though its support was still incomplete.

Challenges

There was an initial ramp up period for the user monitor development team as we had to spend time understanding the flow of control between the various screens, and to understand all of the different patterns and options in how the UI functioned. On several occasions we used case statements as a device to handle the complexity of the GUI.

We found the AJDT integration with Eclipse still sometimes clunky (e.g., needing to restart to allow refactorings) and it would be useful for AJDT to have the “Open Type Hierarchy” feature available for aspects and to include aspects when it is called up for POJOs and interfaces. AJDT still does not do a very good job at visualizing crosscutting structure when applying aspects from one project to others. And because we used load-time weaving for faster compile and test cycles, we further sacrificed effective visualization. Despite these minor issues, AJDT was our primary and most productive IDE for AspectJ development.

We also experienced some challenges relating to the long release cycle of AspectJ 5. Our experiences on other projects as well as this one led us to believe that AspectJ 5 milestone builds after M4 were more stable than AspectJ 1.2.1, and we had concerns

about there being timely bug fixes in that branch if we chose to use it. The ability to use updated Eclipse plug-in tools was another factor that led us to adopt AspectJ 5 early on. However, working with milestone or development builds of AspectJ 5 led to a configuration management problem, primarily related to changing bytecode formats frequently, which requires careful synchronization of the versions of compiler with load-time weaving jars including the versions used in Eclipse when testing with load-time weaving.

For successful project use we believe it will be very important for the AspectJ project to put more effort into supporting branches and to have shorter release cycles now that AspectJ 5 has been released. It's important to understand that AspectJ 5 is integrating support for some major Java language changes, although in retrospect a more incremental approach to releasing these would have been beneficial to projects like ours.

As noted earlier, there were some areas where we needed to expand weaving support for these use cases. The project had recently switched to using Java 1.4 and would have needed compelling benefits to incur the expense of testing and distributing an updated Java 5 VM. At the time we started work, this meant we had to build the non-delgating URL weaving ClassLoader as described before. Even with AspectJ 5's capacity to use a URL Weaving ClassLoader that reads an aop.xml definition file, we still need the non-delegating ClassLoader to integrate easily into IDE launch commands.

The limited ability to visualize AspectJ sources has led to a few occasions where refactorings have broken AspectJ code, which wasn't exposed until the continuous integration build failed. Developers on the project typically assume that if everything works in IntelliJ, then everything works properly, which is a bad assumption for the AspectJ code.

Lessons Learned

Complexity of the Application: The central challenge in designing this solution was to avoid a tight coupling with the existing UI implementation while not making wholesale changes to the UI. For instance, sometimes we had to use a series of correlated pointcuts with `cflow` advice to accurately track processing for wizards. In other instances we needed to refactor the application code to improve modularity and to expose needed data. We used Inter-Type Declarations (ITDs) to manage state about the target classes in certain situations.

Use of Load Time Weaving: During the development and testing of the user monitoring feature we discovered that it was much faster to build our aspects separately and then run with load time weaving to allow testing even in Eclipse where build time weaving is feasible.

Pointcut Stability: One interesting note was that in typical OO programming you try to program with a focus on the parts of the system you anticipate are most likely to change. In our case we had to structure the aspects to depend on design structures that we anticipated would change the least. For instance, if we created pointcuts that were tightly coupled to design structures that are likely to be changed in the near future, then these pointcuts would break, revealing a fragile implementation on our part.

This mirrors our previous experience in applying aspects to any non-trivial subsystem: the hardest part is to fully understand *all* the valid uses of the subsystem. This differs from programming calls from another subsystem which requires only understanding enough valid uses to achieve a specific result. Aspects are invoked by the system, so they need to be correct in all cases. This follows Postel's Law "Be liberal in what you accept, and conservative in what you send." Aspects need to be liberal in responding to any valid means that their collaborators might act, and

conservative in the assumptions, requirements and calls they make in turn. Then again, one of the most important uses of aspects is to modularize interaction protocols, so it shouldn't be surprising that an important rule about protocols applies to them!

Previous Use: The previous effort by the project team to use aspects to capture and playback integration testing ran into the previous two problems: by writing aspects that were privileged and depending on internal implementation details the code was fragile. It would have been better to have refactored the base code to expose important information. Moreover it raised the importance of clear ownership of aspect code: in a sense this is not new, although it raises the importance of awareness and support for developers to work with aspects. We believe the real issue in this previous effort was the low priority of maintaining integration tests.

Policy Configuration: We developed a configuration system that made it easy to allow the system to properly function, even if the aspects couldn't be instantiated. This allowed for a more robust deployment (in case the weave on install failed) and easier development for the developers who weren't using an IDE with AOP support (or who didn't want to incur the extra start up time required for load-time weaving).

Observer pattern: The existing application uses the traditional OO implementation of the observer pattern fairly extensively to propagate information about changes to model objects to the UI and other model objects. We elected to build on a familiar pattern in a context where it works fairly well. Our monitors register as listeners to some existing events, and we also used the existing application pattern to configure the feedback UI as a listener for updates to feedback levels. However, we did use an aspect to observe the different means whereby feedback levels might change (whether initiated by the user or the system), rather than scattering checks in several places. In general, we had to sacrifice the slight benefit of better modularity with aspects to focus on higher value areas with less impact on the project. As the project's familiarity with aspects grows, this is an area where more use of aspects could offer some real benefits.

Organizational Acceptance

The use of a new technology always causes it to be a suspect whenever something goes wrong. In particular, the use of aspects was suspected for IDE integration, for problems in building, or in one case for problems in running tests with code coverage. Because the AOP development occurred in AspectJ, sometimes we neglected to test in IntelliJ. Out of these experiences, we devised the more robust fallback strategy and development from integration jars approaches described above. However, in other cases when there were build problems or test failures, aspects were often suspected when they were not the cause.

The team had been introduced to aspects before and they had adopted them for use in integration tests. However, the previous implementation was not maintained in part because of resistance to regression testing, and in part because the implementation was made without changing the underlying system at all, and as such it was tightly coupled to internal details (e.g., having many privileged aspects), which made it hard to maintain. By contrast, when we integrate feedback assertions in automated system tests, we expect to be able to track problems and to maintain the feedback subsystem.

The project team has been generally favorable to the use of aspects and interested to learn from the data received by the rollout. The two team members who previously developed with aspects are both favorable towards the use of aspects and would like to see them adopted more widely.

For the first phase of the deployment of aspects, most of the project team is developing code that can be nearly oblivious of the use of aspects, so there has not yet been significant interaction or opportunity to apply aspects in other areas of development. We anticipate that the use of aspects for testing will allow more widespread use and will show tangible benefits to the project team as a whole.

Conclusion

Overall, the project has been quite successful, being able to develop an important capability quickly and with minimum intrusiveness. We are looking forward to building on the initial success of this project.

Acknowledgements

Thank you to Ray Tackett, Dan Marus, Sam Konyn, Guangjing Zhou, Andy Barba, Jim Mitchell and the whole NGST team for teaching us so much on this project. Thank you also to Will Edwards for his support on this project, and to Gregor Kiczales for introducing us to the project team.

References

- [FitNesse]A fully integrated standalone Wiki, and acceptance testing framework. http://www.fitnesse.org/
- [Glassbox] Performance monitoring with AspectJ, Part 1: A look inside the Glassbox Inspector with AspectJ and JMX, Ron Bodkin, IBM Developerworks, Sept. 2005 http://www-128.ibm.com/developerworks/java/library/j-aopwork10/
- [Glassbox2] Performance monitoring with AspectJ, Part 2: Extend the Glassbox Inspector with load-time weaving, Ron Bodkin, IBM Developerworks, Dec. 2005
- [LesieckiTest].Unit test your aspects: Eight new patterns for verifying crosscutting behavior. Nicholas Lesiecki, IBM Developerworks, Nov. 2005
- [Ajmock] Virtual Mocking … with jMock, May 2005, Ron Bodkin, http://rbodkin.blogs.com/ron_bodkins_blog/2005/05/virtual_mocking.html
- [StarMOBILE] More information about this device can be found online at http://www.dcctools.com/en/starmobile/

Implementation of AOP in non-academic projects

Allison Duck
Business Objects, Vancouver, BC, Canada
alliduck@gmail.com

Abstract

Aspect-Oriented Programming (AOP) had been actively researched in academia but little has been published about its implementation in industry. The purpose of this study was to investigate the benefits and pitfalls encountered while using AOP in commercial applications. This paper examines the experiences of eleven programmers in five countries who used AOP in their role as project manager, lead architect or developer. Participants were interviewed on a wide variety of topics in regards to their AOP project and the results of those interviews are presented here.

The following observations were made: Interviewees conveyed that AOP was a powerful tool to reduce complexity, defects and workload. The interviewees were early adopters and have implemented AOP in a wide variety of applications. AOP made their code more compartmentalized, readable and reflective of its original intent. AOP suffered common limitations of emergent technologies including fewer resource materials, compatibility problems and minor system complications with earlier versions of the AspectJ compiler.

1. Introduction

This paper examines the experiences of eleven programmers in five different countries who used Aspect-Oriented Programming (AOP) to implement a new application or to work on an existing software program. This paper is written for someone with a basic understanding of AOP. If you are new to AOP there is a wealth of material available on the Web or, for a more comprehensive introduction, Ramnivas Laddads' well-written book "AspectJ in Action: Practical Aspect-Oriented Programming" is highly recommended. This paper is written for those who have either worked with or are interested in working with AOP and would like to know more about other programmers' experiences with AOP.

2. Study Format

1.0. Layout of interview

Participants were asked about the background of their AOP project, the resources they considered useful, the "nuts and bolts" of the project, the actual coding and the results of using AOP. Eleven people were interviewed, all of whom were project managers, lead architects, or developers. All but one interview was conducted over the phone. The other interview was done via email. The duration of each interview ranged from 45 to 60 minutes. These early adopters of AOP were located across the globe: Norway, India, Brazil, Canada and the United States. There were two companies with small groups of programmers who had adopted AOP. The others had single-handedly added AOP to new or existing projects. No one involved in a large AOP project volunteered to participant.

2.0. The interviewees

All but one referral came from AOP experts[1]: Ron Bodkin, Adrian Colyer or Nick Lesiecki. One interviewee responded to a request for participants, which was sent to the AspectJ and AspectWerkz users mailing lists[2]. The self-selected respondents readily agreed to talk in depth about their experiences with AOP. There was no lack of bias here; these were people who, on the whole, were excited about the benefits they had reaped using AOP.

3. Background of the project

The interviewees reported using AOP for a wide range of projects. These include:

- A legacy system migration for a customer.
- Several Web applications including a business-to-consumer Website and a J2EE Web application.

[1] In the vernacular, they are referred to as 'AOP evangelists'.
[2] An AOP expert, Alexandre Vasseur, forwarded this respondent the mailing list request.

- Two different frameworks: one for auditing J2EE enterprise applications and one for creating large-scale business applications.
- Problem analysis using tracing.
- Fine-grained security at the object incidence level for a Swing-based application

The respondents said that they chose their particular projects because of a good match in terms of requirements. They were prominently involved in the project and were able to exercise influence in choosing AOP.

4. Adopting AOP

1.0. Stages of AOP adoption

Gregor Kiczales, who led the Xerox PARC teams that developed AOP and AspectJ, in collaboration with Ron Bodkin, established a three-phase AOP adoption model. A successful AOP adoption must happen in three stages in order to build successive, positive experiences with AOP and to manage risks associated with implementing any new technology. The three phases are:

1.0.0. Phase 1: Introduce exploration and enforcement aspects.

The exploration aspects return information about the system, whereas the enforcement aspects maintain the system requirements for quality control. Both these aspect types are used mainly for development work. This is not to say that these aspects are trivial or ineffectual; these aspects can be quite substantial and powerful as will be examined later in the paper.

This introduces AOP to a team project in a non-invasive way. These aspects are normally transparent to other programmers. One or two developers can create these aspects without impacting the rest of the team.

There were eight interviewees at this phase.

2.0.0. Phase 2: Use aspects to aid in auxiliary/infrastructure software design

After successfully passing through Phase 1, aspects are used to implement system infrastructure such as exception handling, transactions, threading, synchronization, caching. Aspects may or may not be transparent, depending on how they interact with the system.

There was one interviewee at this phase.

3.0.0. Phase 3: Make AOP core to business code

Aspects form the core of the system's functionality. They are now a crucial part of the design. Other developers are aware of the aspects and how they impact the code. Aspects are a central part of the main code and, therefore, not transparent.

There were two interviewees at this phase.

2.0. Why AOP?

There were many reasons why AOP was chosen. As one programmer succinctly put it, "After searching, [AOP] was the best solution." Another said, "[AOP] made conceptual sense." Other factors stated were:

- productivity
- quality/consistency
- ease of modification of code
- transparency

Some interviewees chose AOP as it made conceptual sense; for others, it helped avoid perceived limitations of expressiveness in Java. Some went for AOP because of the ability to modularize concerns. Still others decided on AOP because it improved infrastructure code. Some chose it simply because it was the best solution in terms of transparency, consistency, and ease of use across applications or between application tiers.

3.0. Capacity in the project

The participants had strong programming backgrounds and held senior programming roles - most were senior developers, chief architects, team leads or project owners.

4.0. Number of AOP programmers on the project

The number of people programming in AOP was usually one or two with a maximum of four. Of the people interviewed, five were the sole AOP programmers, two were in a group of two and four were in a group of three or four.

5.0. Number of other non-AOP people involved

The number of other developers involved in the project ranged from 3 to 15. The ranges were split fairly evenly into these five categories: 3 other developers, between 3 and 6 developers, between 8 and 10 developers, greater then 15 developers, and varied (one was a consultant, another did project support).

6.0. Person who programmed core code also programmed AOP

The majority of the solo (or near-solo) AOP programmers worked on legacy code and wrote aspects to help them. Only two of the solo (or near-solo) programmers and one of the groups of AOP programmers developed core code and wrote all aspects. Only one group developed all the core code and wrote all the aspects. At the time of the interview, that project is still ongoing and several more developers have been added to it.

7.0. Amount of time spent programming with AOP

The amount of time spent programming with AOP in general ranged from 1½ to 2½ years. Typically, the aspects were programmed during an intense work period, which varied from several weeks to an entire one-year project development.

8.0. First exposure to the concept of AOP

The majority of study participants were first exposed to the concept of AOP by reading the October 2001 version of the "Communications of the ACM (IEEE)." Another group attended a presentation by Ron Bodkin on AspectJ and AOP at the *No Fluff, Just Stuff* lecture series and knew immediately this was a solution to their problem. In one case, an in-house expert explained it to them in an "immediate way." He was available to answer questions as they naturally arose. The rest of the programmers had either read about AOP online at TheServerSide.com or other Web sites, heard about it through colleagues, or could not remember.

5. Resources

Nine out of eleven interviewees cited Ramnivas Laddads' *AspectJ in Action* as the most useful AOP resource book. One programmer learned AOP by working through this text in 48 hours straight. Another said he was eagerly awaiting Laddads' next book on AspectJ 5.

There were two people with differing opinions of Adrian Coylers' book. One highly recommended it because it "motivates the tools and the tools help motivate the concepts." Another disliked it as he felt it was "written for a mixed audience: very basic [to begin with], for those with limited exposure to Java programming and then [it] skips to advanced compiler details."

Other highly recommended resources were two blogs; Ron Bodkin of New Aspects of Software (rbodkin.blogs.com) and Adrian Coyler of IBM (www.aspectprogrammer.org/blogs/adrian). One programmer said that while there were "several blogs which talk about [AOP] in interesting ways, [Adrian's] is the one that I've had the most 'Ahas' with." Websites that were recommended repeatedly were the Eclipse resources for AspectJ (www.eclipse.org/aspectj) , and for AspectJ Development Tools, AJDT (www.eclipse.org/ajdt) and TheServerSide.com.

When people were stuck on a problem that could not be solved by discussing the problem with a fellow programmer (or beating their head against the wall, as one solo AOP programmer suggested), they would turn to either the AspectJ documentation or the AspectJ users mailing lists archives. If that search didn't return a solution, they would send out a message to the users group mailing list.

1.0. Training on AOP

Most people weren't trained formally; they expanded their knowledge of AOP by using it. In order to master the technicalities of aspects, such as pointcut syntax, the programmers learned by working with the aspects and the code. "Playing" is how more than one developer put it. A couple of people had downloaded the AspectJ compiler and worked through some of the given problems. Still others worked through the "AspectJ in Action" book (one in 48 hours as previously mentioned). Several people learned about AOP through either in-house or external presentations.

2.0. "I wish I knew _____ about AOP before I started."

The interviewees were asked to fill in the blank in the sentence; "I wish I knew _____ about AOP before I started." The most humorous answer was "more." Others said they wished they knew when to use AOP. A few said they wished they realized sooner that AOP was not a "golden hammer", a solve-all for any programming dilemma. They said, in retrospect, they were so pleased with the results of using aspects that they had tried to use aspects where they didn't apply. Several said that they wish they knew about the bugs in the tools, most of which have now been fixed, before they began working with AspectJ. For instance, the incremental AspectJ compiler on older versions of Eclipse had tooling issues. This will be discussed in more detail later in this paper.

6. Nuts & Bolts

1.0. Criteria for aspects

The main criterion for aspects was that they be robust. In one programmer's case aspects had to have the ability to switch from being general to being specific. In another instance, the aspects had to be "generic enough that [they] wouldn't need to be modified for other applications." Transparency was also a concern; aspects needed to be the invisible hand, affecting the code without being obtrusive. As one participant put it, "Above all, the program should have no noticeable changes to the end user if aspects are on or off. The functionality should be the same." [3] There were a couple of technical requirements, such as "must use load-time weaving."

2.0. Types of aspects used

The most common type of aspect used was the enforcement aspect, followed by development process aspects and exploration aspects. One project also used core logic aspects and infrastructure aspects. Another project used both core logic aspects and development process aspects.

Enforcement aspects allow the programmer to define architectural and coding style rules and ensure they are followed consistently in the code. A common usage was to find and convert system.out.println into debug statements. Another enforcement aspect created an error if files were saved using the wrong letter case. Another enforcement aspect helped high-level customer support diagnose problems in complex sections of the customer's code.

Exploration aspects helped an auditing software application to investigate what method calls were performed on EJB (Enterprise JavaBeans), JDBC (Java Database Connectivity), or SQL prepared statement execution. The aspects were used against J2EE interfaces. Using AOP gave the code auditors an immediate, meaningful view of what was actually going on in the system. They had better insight into the code than could possibly be done by just looking at the raw code or flat output. Using AOP allowed them a more complete view of the inner machinations of the audited system. Programming exploration aspects was less laborious than, for example, adding debug statements to each and every method.

An illustrative example of the power of exploration aspects is the story one programmer told of being hired on to find out the root cause of unusual runtime behaviours in a Web application. After some consideration on how to approach this problem, he solved it by writing a ten-line aspect to follow the thread execution throughout the program, which ended up saving him time and many, many lines of code.

The development process aspects were most often used to log information. In one case, the *exception introduction pattern* for performance profiler as described in *AspectJ In Action* was used. Logging and tracing were also used. One participant had created a virtual mocking framework, which tested the system.

Most of the aspects were exploration, enforcement or infrastructure type aspects. This may have to do with apprehension on the part of the non-AOP programmers of integrating core logic aspects into the system as AOP is still considered by some to be "bleeding edge" technology.

The aspects themselves were split almost evenly between advice and intertype declarations.

3.0. Range of scope

The majority of applications had a large range of scope; they crosscut most classes. The application with the second largest scope crosscut at most 200 persistent classes. The application with the largest scope crosscut about 700 classes, with a few thousand methods. The range of scope was flexible for the applications that used infrastructure aspects. In those cases, the range varied depending on the application it was being run against. According to one developer, "If an application had a lot of EJBs then it would pointcut a lot of classes. It really depended on how many different J2EE interfaces that it was pointcutting against. So the bigger the application, the more things we would pointcut." Only one project had a small range, with very few classes involved, as the company was sold and hence the application while it was in the prototype stage.

4.0. Homogenous or heterogeneous aspects

A homogenous aspect applies consistently across the code wherever the pointcut is matched. Logging and tracing aspects are examples of homogeneous aspects. A heterogeneous aspect impacts multiple places yet exhibits different behaviour in each place [4]. An

[3] This as striking as AOP can be used for much more than this, according to AOP evangelists.

example of a heterogeneous aspect is an aspect that decides between two different protocols.

Eight of the eleven programmers solely used homogenous aspects for logging and tracing. As one interviewee said, "[The aspects are] very simplistic; in fact they're very thin. There isn't much code in any of the aspects."

One of the developers used solely heterogeneous aspects because his aspects made decisions based on "architecture, technology platform, and design decisions" for a model driven development framework.

The other two developers used a mix of homogeneous and heterogeneous aspects. In one case, the heterogeneous aspect made a decision "to build an EJB-based system or use local execution instead." In the other case, heterogeneous aspects were used for making decisions about exception handling.

5.0. Modeling and designing aspects

None of the participants used formal or high-level notation such as UML diagrams. The AOP design process was informal and iterative. The majority of interviewees used whiteboards. Two of the solo AOP programmers just programmed into a text editor or their IDE and ran the code from there. The whiteboard was used as it was better for "emergent design" and the developers got a quick visual idea of what was going on. Generally, they used the whiteboard to write down some sample cases and data flow. Interfaces were sometimes sketched out and methods named; groups were captured and diagrams drawn. After working on the whiteboard, they would try out their ideas, see how they worked and then go back to the whiteboard to repeat the process until completion. Regardless of design technique, aspect creation was an iterative process.

7. Implementation

1.0. Language/extension used

All of the code was written in Java. Ten out of the eleven programmers used AspectJ. One group had originally written their application in AspectJ and then had to switch to AspectWerkz in order to be compatible with the target platform, BEA WebLogic. However, the project manager said now that AspectJ and AspectWerkz have merged and "there's going to be some runtime capabilities in AspectJ, we're going to be going back [to AspectJ]." One solo AOP programmer initially programmed in AspectJ, and then for compatibility issues had someone else write it in AspectWerkz. There was one programmer who worked solely with AspectWerkz. Another programmer was in the initial stages of programming in Ruby as he felt it was more suitable for AOP.

2.0. IDE used

Eclipse was used by all but one programmer who used NetBeans for AspectWerkz.

3.0. Implementation Testing

Most people said that implementation testing with aspects didn't differ too much from regular implementation testing. Some people integrated independently tested classes and aspects and others did "big bang" testing. Overall, implementation testing seemed to go fairly well. However, according to a couple of people, the challenge came in "trying to automate those tests." One said "integration testing worked well, no problems, pointcuts [were fine], if it worked in one case [it seemed to work in all]." One person said they found their experience "very satisfactory."

4.0. Debugging in AOP

Generally, those interviewed found debugging slightly more difficult than normal. They said that AOP was more time consuming to debug. One interviewee said that he "found debugging a little bit challenging for the first 5 minutes. Eclipse is perfectly capable of debugging aspects, you can put a breakpoint in your aspect and it will stop in the proper place. If you have a pointcut related to a method call or an accessor, at first it's a bit weird but then you realize you can just play through them. We had some tricky problems and had to debug them but it was because they were tricky, not because of the debugger." He also said, "I'm still searching for a really good way that doesn't take a ton of effort to test on aspects to make sure the pointcuts are matching where you're expecting them to, hopefully not matching on things you're not expecting them to."

Two interviewees mentioned that they had difficulty with debugging and weaving in AspectJ. One said that the compiler would stop weaving if there was an error. However, this problem has now been addressed in version 1 .2.1 of AspectJ[4]. The other said that the error

[4] http://www.eclipse.org/aspectj/doc/released/changes.html 74245 Specifying the -proceedOnError flag will now cause the compiler to attempt weaving even in the face of errors.

messages returned while weaving were not clear. This has been fixed in version 1 .2 of AspectJ[5].

One interviewee had trouble with the lines not matching up. This seems to have been fixed in the 1.2.1 version of AspectJ. Yet another interviewee mentioned that, in Eclipse, if the compiler "inline" flag is set to "no", the line numbers are valid and the debugger can ascertain where it went wrong. This makes it "much, much easier" to debug an aspect. One person mentioned "having clear pointcut naming helped with debugging."

One participant stated that they felt frustrated because AJDT doesn't provide an eager parser so the outline view would not be in sync. When he changed code in the editor, it would take a full rebuild before the outline view was updated. Using Eclipse for non-aspect classes, the parser is eager so he was accustomed to seeing his changes immediately. Because of this, he had a hard time understanding what was happening where. As a workaround, he wrote the aspect in a text editor, compiled and ran the application with print statements where aspects would be. If everything was as expected, he would activate the real code after that. This issue has been acknowledged on the eclipse.org Website[6].

8. Results - Qualitative

When asked, "Would you use AOP again?" the answer was a resounding "yes." The interviewees said they would use it in their own projects if certain conditions were met and indicated it would be particularly useful for large projects with crosscutting concerns. One person said that the version of Eclipse would have to be greater than 1.2 and that they would use it on a "small team, with modern tools, good communication and a high level of expertise", which is a good description of the two teams that successfully used AOP. Another person cautioned that aspects are very powerful and could be "dangerous" if misused to create, for instance, a situation where overmatching occurs. Another person stated that they now have automated aspects making it easier to use AOP if a project called for it and was large enough.

Participants were asked to give three advantages and three disadvantages of using AOP.

1.0. Advantages of AOP

The main advantage, epitomized in the words of an interviewee, is that AOP "does exactly what you want." And it does it quickly, cleanly and powerfully according to the interviewees. AOP reduces complexity, defects and workload. Other stated benefits of applying AOP were:

- AOP works. There's no esoteric reason why these programmers are creating and implementing software using AOP. They are using AOP because it helps them achieve programming goals. This was the number one reason stated for using AOP.

- AOP is powerful. Another reason stated for using AOP is that its strong design ability helped improve productivity. For one programmer, it was the ability to achieve good results with a few lines of code. AOP helped them move towards having one design concept to one implementation.

- AOP keeps code cleaner and cleaner code is an advantage: the core code looks "elegant" and this leads to improved readability. As applications are continually revised and programmers work on code that they did not write, it is advantageous to "[see] what the developer was thinking."

- AOP saves time. One developer sold his group on AOP by pointing out that "if it takes 1 developer 6 minutes per class and we can use an aspect instead …" A few programmers mentioned the speed of aspects and how quickly an aspect could be written. Aspects not only saved time, they also reduced code size. One programmer's experienced this fully after switching to another project that didn't have aspects to do infrastructure support. He said "You realize all of the heavy lifting that AspectJ was doing…All of those things added up to when you're writing code in a main project …is just a lot of extra stuff to think about [that you'd rather not think about]…You tend to forget that very quickly until it goes away and then you find it again."

- AOP makes changes easy. Because modifications are confined to the aspects themselves, programmers do not have to search through hundreds of lines of core code to make the change. The aspect alone need be changed. This helps keep the code consistent and "avoids tedious retyping of code."

- AOP encapsulates modular concerns. Several of those interviewed liked the modularity, the ability to code a crosscutting concern in a single place. Modularity is one of the central tenants of OOP.

[5] 54819 Error and warning messages coming from the weaving phase of compilation now show source context wherever it is available, and also indicate as the source location of the error either the class file or jar file from which the binary source unit came.

[6] http://dev.eclipse.org/viewcvs/indextech.cgi/ajdt-home/faq.html?rev=1.3#q:outlineupdate

However, as one interviewee stated, "Object-oriented programming is good but it has limitations; it can't solve crosscutting issues easily."

- AOP helps manage complexity. One participant said, "AOP is one of the tools for managing complexity. And complexity is the death nail of big projects over time. AOP decreased the complexity."

- AOP can be used transparently when necessary. One mentioned that they liked the fact that AOP could be used transparently to hide the detail from those not familiar with AOP. The advantage of reusable and flexible aspects was also mentioned.

- Two programmers mentioned how AOP gave them the "power to redefine how parts of language work." They said they liked being able to extend the language.

2.0. Disadvantages

The disadvantages of using AOP varied greatly. There were issues with past versions of AspectJ and Eclipse. As one interviewee said, "Historically, debugging has been a problem. Even just understanding how things apply" can be a struggle for someone new to AOP. Some disadvantages cited were:

- Length of time to compile. One complaint was that the AspectJ version 1.1 took too long to compile. One of the teams started with AspectJ, version 1.1 but ended up switching to AspectWerkz as it had load time weaving. The other team was locked into using Eclipse 1.2 because they were using WebSphere Studio Application Developer. They added in "-proceedOnError", the development build of AJDT (1.1.6), and ANT script to use AspectJ 1.2, which helped. He said that they were still having many problems with "proceed on error" weaving and compiling. These should have been solved with the AspectJ 1.2 release. According to the Website, compared to AspectJ 1.1.1, AspectJ 1.2 is "faster, with weaving completing in less than half the time it used to take in many cases."

- Less AOP than OOP resources. A common disadvantage cited was that, as AOP is a relatively young concept, there is not the wealth of information to draw upon as there is with OOP. A Google search for a problem in AOP is not going to return the number of hits as for a similar issue in OOP. Another complaint was that "robust aspects systems" were limited to Java. A C++ programmer is going to have a drastically smaller subset of information to draw upon for AspectC.

- Aspects as "red herring" as mentioned in Nick Lesiecki's article[7]. Aspects are often blamed for any problems that arise. A few people mentioned that if an aspect was involved in a system, it was the first thing blamed, regardless of the truth of the situation. One example that was given was when working with a customer's (prototype) code, a null value was passed in as an argument. A NullPointerException was thrown and when the stack trace was examined, the aspect appeared prominently. It looked to the untrained eye as if the aspect broke the code. The developers turned this experience into a principle: "we realized we really need to take a good look at the aspect and try to account for all scenarios that might occur so we don't look like we're breaking someone's application." They had to make their aspects rock-solid so AOP wasn't blamed instead of unmasking the true problem. One interviewee warned "make your aspects bullet-proof."

- AOP tools were not always compatible. One developer said "as AOP is not widely adopted, [one] cannot assume [other AOP tools] will be compatible. For example, [we] had problems with compatibility for JVMs, such as JRockit."

- AOP training is required. Not a lot of developers know about AOP, however, from previous comments, the take-up time of AOP for a skilled OOP programmer seems to be reasonably small as opposed to shifting to OOP for a functional programmer. There was apprehension about unskilled programmers using AOP. One person was concerned about the potential problem of a widely matching aspect that was "woven left and right", everywhere in code (overmatched). However, this comes down to expertise in programming; an inexperienced, poorly taught programmer is a danger regardless of programming language or methodology.

- Not being able to see where the code matches before compiling. One developer said, "It would be useful to have some tool…to be able to take a set of application code and just have a pointcut and to be able to analyse that code and lift out all of the places that the pointcut would have matched. Just quickly write a pointcut in a window and it shows what code in my whole project that would match.

[7] http://www.aosd.net/2005/archive/Applying-Aspectj.pdf

[This] would be useful for challenging aspects. For example, ones that matched too much. As challenging to not match the wrong thing as it is to match the right thing."

- Inability to do a "hot swap." One interviewee lamented not being able to do a "hot swap." He said that in a "hot swap … the developer is able to make changes to a class which is loaded in a running application, and the class will be automatically swapped out for the modified version without stopping and restarting the application. When running in a full J2EE environment with servlets and EJBs, starting up the application can take several minutes, and it may take several more minutes to set the state and navigate to the page you are testing. So being able to make changes to code and have it take effect immediately without stopping and restarting can be a huge benefit. The standard JDT provides this, so developers expect this in the AJDT as well."

9. Quantitative Results

The question, "What metrics did you take?" had a universal reply: a short burst of laughter, and a single word, "none." These developers were intimately familiar with their code; they knew that with aspects it worked better and faster. "[We had] no formal review," said one, "[it was] absolutely apparent that it was better." If a performance problem occurred, they would address it immediately or remove the aspect. When problems did occur, such as an aspect under- or over-matching, teams would deal with this informally adjusting the aspect as need be. Most said they didn't really put any value in measuring lines of code. One group did informally run metrics using Maven (software).

The developer who had a negative experience due to being tied into an older version of AspectJ, said that using aspects "reduced [time] in general, 2.5 person months but the extra compilation time blew that out of the water."

10. Interviewees As "Early Adopters"

In *Crossing the Chasm*, Geoffrey Moore divides the people who accept technology into five categories. The interviewees all seemed to fall into the second category: early adopter. Characteristics of early adopters include:

- Articulate
- Pragmatic
- Willingness to experiment
- Skillful

1.0. Articulate – able to sell their ideas

The developers and project managers who embraced AOP were enthusiastic and articulate. They spoke with ebullience about their programs. They were thrilled about how clean and modular the code was. They were convinced of the power of AOP to manipulate the code, that it gave them, as in the words of one interviewee, "[the] capability to do things that aren't part of the language." Their passion for AOP was persuasive. After speaking with them, it would be hard to not be excited about AOP.

2.0. Pragmatic – want to solve a real problem

The interviewees were not starry-eyed believers; they had a healthy skepticism of AOP and were aware of the potential pitfalls of viewing AOP as a "golden hammer." They had clear expectations of what they expected AOP could and could not do for them.

3.0. Willingness to experiment - whatever will do the job

They reported that the creation of aspects was iterative – the creation process was informal with a lot of communication back and forth. Interviewees shared the common characteristic of tenacity, the ability to keep going until a solution was found. As these are the AOP pioneers, there was less reference material available when stuck on a specific predicament. For example, as two separate developers mentioned, a Google search on an issue in AOP will produce fewer results than a search on OOP. None of them had a clear path to follow. But almost all were further ahead than if they had stuck to OOP instead of using AOP in regards to productivity, code clarity and reduction in errors.

As one developer said "To do AOP well you have to stop thinking of it as special solution, think of it as any other tool in your toolbox and not be afraid of using an aspect. If it's a crosscutting concern and it doesn't seem like that big a deal, you shouldn't be afraid to use experimental technology."

4.0. Skillful – high level of expertise

The interviewees were adept programmers who were fluent in technical terminology. All of the interviewees seemed comfortable with using an amalgamation of applications. They freely mixed multiple commercial and open source applications. They felt comfortable taking things apart, for example, going into the library of an open source code application to change something

unsatisfactory. They ended up doing amazing workarounds rather than giving up. Even though there was often not a clean, obvious solution to a problem, they were not thrown by the complexity of the situation. This might be accredited to their advanced level of programming expertise but it appeared to be more of a common attitude rather than a measurable amount of experience.

11. Explaining AOP To Others

1.0. Easiest idea to convey

Almost all those interviewed suggested giving quick conceptual examples as the best way to explain AOP, several used tracing as an example. Instead of writing trace statements for each method under inspection, a single aspect could be written with the methods of interest designated in the pointcut. One person said that tracing was such a powerful explanation of AOP that people's eyes glazed over with the thought of the possibilities. The problem was, he said, some people walked away thinking that tracing was the only thing that could be done with AOP. Another person recommended "showing the classic example … of logging in Apache, [how the aspect] goes across many classes and the code is just in one place. When people see the aspect and how short it is, that's easy to understand." Another said that "the application client/server is the best way to explain AOP." Regardless of the chosen example, people had an immediate understanding when shown how a particular concern was implemented by a corresponding aspect.

Other ideas that were quickly picked up on were how aspects aided in clarity by not cluttering the code, the concept of modularity and the concept of instantiated (or realized) interfaces. One person said, "People from a C++ background can understand the idea of multiple inheritance immediately." Another interviewee said they had heard it described as "coding on a 'z' axis. If existing code is on flat paper, [AOP] can come in from above and apply little bits here and there."

Other people said they had the most success when building on what the person already knew, putting it into their syntax. For example, one interviewee explained AOP in terms of Java syntax; an aspect is like a class and a join point is like a method signature.

2.0. Challenging ideas to convey

The challenge came when explaining applications of AOP past straightforward applications, for instance, moving to using AOP for production or core business logic rather than infrastructure support. But one of the pitfalls of not doing this is that if AOP is explained in the context of logging, people will say, "Isn't AOP just for fixing bad design?" Or according to another interviewee, "If you only show simple examples in AOP, people are going to say, why not just use OOP?" To get around this, one interviewee said that once he explained "AOP can only do what OOP can do, just as OOP can only do what functional programming can do" people understood the role of AOP better. He would use simple examples to explain the concepts of AOP and then finish his explanation with his future vision of AOP, for instance, moving on to create architectural aspects.

As one developer eloquently said, "There's so much subtlety within the language and yet very few keywords which makes it difficult to convey how much power it adds. It's easy to explain simple uses, the simple syntax; it's a bit more difficult [to show how AOP can do so much]."

One said that the biggest barrier to understanding was answering, "Why do we even want to do this?" A concept like "join point" is easier to understand once people know the benefits of modularity that AOP provides. Terminology follows understanding; keywords only become useful once AOP is broadly understood.

That said one point of confusion was that "[join] points and pointcuts are different things but sound like the exact same thing. Better naming would be more helpful." New learners seem to have trouble differentiating between join point and pointcut.

A few people said that they struggled explaining the syntax of pointcut declarations. One (bravely) admitted that this area was "troublesome [to him] so [he] struggled to explain to others." Specifically, two areas of pointcut syntax that were difficult to explain were "call" versus "execution", and "this" versus "target."

As expected, some said that advanced topics like inter-type declarations (introductions) and mixins could be confusing to explain.

3.0. Ideal communication setup

Developers who work along side each other in close quarters had an easier time passing on their knowledge and enthusiasm about working with AOP. The most challenging one was a distributed, trans-national team. This developer had less than ideal results with applying

AOP, which may have been due to not following the phased adoption process. Phase 2 aspects rather than Phase 1 aspects were used opaquely which resulted in aspects being blamed for problems in the code that it did not cause.

12. Conclusion

Almost all of the aspects used by the interviewees were for Phase 1 (exploration and enforcement aspects) or Phase 2 (infrastructure aspects) of the AOP adoption sequence. Those interviewed were, in general, keen supporters of AOP. They understood that AOP aids modularity and encapsulates crosscutting concerns. They were using AOP because it helped them achieve programming goals. Its strong design ability helped them improve productivity and achieve results. In addition, AOP kept the code cleaner and reduced code size. The interviewees found AOP encapsulates modular concerns, helps manage complexity and can be used transparently when necessary. AOP was also timesaving, from the speed of creating an aspect to the modifying the aspect directly rather than going through many lines of core code. This also decreased the possibility of errors through retyping of code and kept the code consistent.

Some interviewees expressed frustration with the limitations of past versions of AspectJ and tools such as AJDT, for example, compile time and lack of compatibility with other tools. Many of these issues have now been solved, and are continuing to improve as the tools evolve and newer versions of AspectJ are created. The issue an eager parser in AspectJ is still unresolved.

Another disadvantage cited was that as AOP is relatively young, there is not the wealth of information to draw upon. However, the body of knowledge about AOP has grown since these interviewees first began working with AOP. The interviewees were the early users of AOP. Since they began, improvements have been made to the tools and language extensions such as AspectJ. As more people are becoming familiar with AOP, the information and available knowledge grows. AOP is gaining momentum: for example, the University of British Columbia has hired AOP experts to research and disseminate information on AOP; it is being taught in some undergraduate courses and AspectJ study groups are being created. Although specialized training is still required, a skilled OOP programmer can learn AOP relatively quickly.

Overall, the interviewees were happy with the results of using AOP and would definitely use it again. They willingly shared their experience and knowledge regarding AOP. They were enthusiastic about AOP, as they had found it to be a useful tool in their programming.

13. Future Work

Some possible future studies include repeating the study with a larger group, examining failed implementations of AOP and investigating large software projects that have implemented AOP.

14. Acknowledgements

This paper was originally developed at the University of British Columbia under the supervision of Gregor Kiczales. Thanks to Gregor for his wisdom and time and the reviewers of this paper for their insightful comments.

15. References

[1] Adrian Colyer, Andy Clement, George Harley, and Matthew Webster. *Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools*. Pearson Education, 2005.

[2] Tzilla Elrad, Mehmet Aksit, Gregor Kiczales, Karl Lieberherr, and Harold Ossher. Discussing aspects of AOP. *Communications of the ACM*, 44(10):33–38, 2001.

[3] Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Aks¸it, editors. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2005.

[4] Boris Jabes. Aspects in the Real World. *CMU CS 15-819 Objects and Aspects: Language Support for Extensible and Evolvable Software.* Nov. 13, 2004. Sept. 2, 2005. www.cs.cmu.edu/~aldrich/courses/819/slides/aop-middleware.ppt

[5] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. Getting started with ASPECTJ. *Communications of the ACM*, 44(10):59–65, 2001.

[6] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning, 2003.

[7] Geoffrey A. Moore. *Crossing the Chasm: Marketing and Selling High-Tech Products to Mainstream Customers*. Harper Collins, New York, 1991.

[8] Gail C. Murphy, Robert J. Walker, Elisa L. A. Baniassad, Martin P. Robillard, Albert Lai, and Mik A. Kersten. *Does aspect-oriented programming work? Communications of the ACM,* 44(10):75–77, 2001.