Distributed Systems Development: Can we Enhance Evolution by using AspectJ?

Cormac Driver Siobhán Clarke

Distributed Systems Group,
Computer Science Department,
Trinity College Dublin,
Ireland
{Cormac.Driver, Siobhan.Clarke}@cs.tcd.ie

Abstract. Problems relating to modularity result in the under-performance of the object-oriented software development paradigm in a number of areas. Aspect-oriented software development (AOSD) is a relatively new technology that extends modularisation capabilities in computer software. In particular, *crosscutting* concerns can be modularised. A crosscutting concern arises in a software system when the implementation of a system requirement impacts on more than one implementation module. Such a property hinders the ease with which the software can evolve. With AOSD techniques, the ability to modularise crosscutting concerns results in software that exhibits greater evolvability, as it enhances changeability, pluggability and comprehensibility. This paper reports on the impact on system evolvability arising from the re-implementation of an existing object-oriented system using AOSD techniques. In particular, AspectJ is used, which is an aspect-oriented extension to Java™.

We found that the use of AOSD techniques and AspectJ™ can greatly enhance the modularisation of certain concerns, leading to enhanced evolvability properties. However, for other types of concern the effects on evolvability were less positive. The difference between the two types of concerns related to the extent to which they could actually be modularised using AspectJ.

1 Introduction

Software engineers have long been aware of the need to make software less complex, with evolvability being a major benefit of complexity reduction. A concept that is fundamental to reducing overall complexity levels in computer software is *modularity*. By keeping related modules together, and separating them from modules addressing unrelated issues, software systems can evolve freely without establishing restrictive dependencies. The practice of dividing different areas of interest into separate, independent modules is referred to as *separation of concerns* [1]. It has been established that total separation of concerns is not possible with the current standard programming paradigms [2]. Object-oriented programming is the current standard paradigm for software development. Promoters of the approach claimed that it can fundamentally aid software development by creating architectures that better fit with domain models [3]. While this is true, the whole story is not being told. There are many software development problems to which a suitable solution cannot be achieved with object-orientation. Problems relating to modularity result in the underperformance of the object-oriented model. Object-oriented code suffers from two phenomena known as code *scattering* and code *tangling* [4]. Scattering is evident when similar code is distributed throughout many system modules, with the risk of misuse and inconsistencies at each point of use. Tangling occurs when two or more concerns are implemented in the same implementation module (most commonly the object-oriented class or method), making the module harder to understand and change. There is a need for a new software development paradigm to address these shortcomings.

Aspect-Oriented Software Development (AOSD) is a relatively new technology that extends modularisation capabilities in software development [5]. The AOSD community propose that it is possible to modularise the crosscutting aspects of a system using AOSD techniques. In this context, a crosscutting aspect can be thought of as a requirement in a software system that affects more than one implementation module during its implementation. Applying AOSD techniques can lead to a system that exhibits cleanly captured concerns within its codebase and possesses numerous beneficial properties, most notably evolvability.

This paper reports on our experience re-implementing an existing object-oriented distributed software system using AOSD techniques and the AspectJ™ programming language. The AspectJ version is assessed against evolvability properties such as changeability, pluggability and complexity. We conclude that the extent to which evolution is affected is directly related to the type of concern that is implemented. The remainder of this paper is organised as follows. Section 2 provides background information on the re-implemented software system, AppTrack, as well the aspect approach to AOSD and the AspectJ programming language. Section 3 presents two crosscutting concerns, identified in the object-oriented version of the AppTrack system, that were re-implemented using AspectJ. Section 4 discusses the impact that the aspect-oriented re-implementation had on the evolvability properties of the system. Section 5 presents work related to ours. Finally, section 6 presents our conclusions.

2 Background

2.1 AppTrack

AppTrack is a web-based distributed information system that was developed by a team of programmers from the Computer Science Department, Trinity College Dublin. The fundamental system requirement was to automate the department's postgraduate applications system, making it possible for students and department staff to administer postgraduate applications online. The system was written using Java™.

AppTrack follows the standard three-tier architecture for data-driven web-based applications. The application code was written using the Struts framework [6]. Struts is an open-source framework from Apache's Jakarta project for building web-based applications. The framework encourages application architectures based on the Model 2 approach, a variation of the classic Model-View-Controller design pattern [7]. Users interact with the system via a web browser displaying Java Server Pages (JSP). The information gathered from the user via the forms on these pages is represented in the application code by Struts-defined `Form` objects. Each `Form` object is associated with a Struts `Action` object and the form information is used when the associated action executes. Each `Action` object contains the core business logic related to a piece of application functionality. The `Action` classes that make up the `action` package use classes from the remaining packages (`bean`, `mail` and `db`) to carry out their required behaviour. Once an action has been taken, the `Action` class redirects the system user to the appropriate JSP.

The use of the Struts framework enforces design and implementation rules on the application code i.e., classes must implement specific interfaces and follow certain naming conventions. While conformance to the Struts framework leads to a good degree of modularity, there remain some concerns (non-MVC related) that are not cleanly separated. AppTrack is composed of 81 Java classes and 20 JSP.

2.2 Aspects and AspectJ

The aspect approach to AOSD was first proposed in [2]. A system property or requirement to be implemented is an aspect if it cannot be cleanly encapsulated in a component that is well localised, easily accessed and composed. The goal of aspect-oriented programming (AOP) is to support the programmer in cleanly separating components and aspects from each other, by providing mechanisms that make it possible to abstract and compose them to produce an overall system. This goal is achieved by adding an *aspect language* and an *aspect weaver* to the set of standard tools used in the implementation of a software system. The aspect language is separate to the component language (Java, C++ etc.) and is used to program aspects. The aspect weaver is used to integrate aspects written in the aspect language with the rest of the program code, which is written in the component language. The aspect weaver is a tool that accepts both component and aspect language as input and outputs the combination of the two in pure component language. AspectJ [8] is the most prominent and widely adopted [9] programming language supporting the aspect approach.

AspectJ is an aspect-oriented extension to Java that supports general-purpose aspect-oriented programming [10]. The word ‘aspect’ has a dual meaning in AspectJ, with the meaning depending on the context in which the word is used. At the design level an aspect is a concern that crosscuts (as previously described). At the implementation level it is a programming language construct that enables concerns to be implemented in a modular fashion. This programming level aspect is the dominant unit of decomposition in the AspectJ programming language.

AspectJ is based on a concept referred to as the *dynamic joinpoint*. Dynamic joinpoints are points in the execution of a Java program e.g., method calls and class attribute access. The joinpoint is the foremost new concept that AspectJ adds to Java, with the majority of the rest of language being made up of simple constructs based around this notion. *Pointcuts* and *advice* dynamically affect the program flow whereas *introduction* statically affects a program’s class hierarchy. Pointcuts select certain joinpoints (and values at those points). Advice is code that is executed when a specific joinpoint, identified by a pointcut, is reached. Advice can be run before, after or around a joinpoint. This is the dynamic behaviour of AspectJ. AspectJ uses introduction to statically modify a program's static structure. The members of classes and the relationships between classes may be modified by introducing new member variables and methods into a class or by either defining new parents for an existing class or defining that a class implements another class. Analogous to the class in object-oriented programming, the aspect is AspectJ's unit of modularity for encapsulating crosscutting concerns. Aspects are defined in terms of pointcuts, advices and introductions.

3 Separated Concerns

The presence of crosscutting concerns in an application codebase greatly affects the ease with which the application can evolve. Separating concerns is fundamental to achieving a suitable level of modularity so as to enhance system evolution. AspectJ aims to aid concern separation by providing explicit mechanisms to modularise code.

Following a manual investigation of the AppTrack codebase a variety of crosscutting concerns were identified. These ranged from development concerns that implement house-keeping duties such as tracing and enforcing coding standards, to production concerns implementing core application logic such as exception handling and database transactions. Due to spatial limitations not all concerns can be discussed in this paper. The following is a full list of the concerns that we identified, re-implemented and assessed: Tracing, Transactions, Enforcing Factory Design Pattern, Database Compatibility, Design by Contract (Preconditions), Exception Handling, and Recording Bean Properties Modification. This section discusses the aspect-oriented implementation of two of the most prominent concerns in the AppTrack codebase, one a development concern (tracing) and the other a production concern (transactions). We choose to discuss these concerns as their re-implementation is most representative of the way in which AOSD with AspectJ can affect the evolvability of a software system.

3.1 Tracing

During the development of the object-oriented version of the AppTrack system the development team predictably found themselves in situations where their expectations had not been met by a certain piece of functionality that they had implemented. Non-conformance to system requirements manifested itself as errors ranging from minor bugs to major deficiencies. Although following a test-first [11] approach to system development, they sometimes lapsed back into the classic scenario in which they found themselves inserting printline method calls into the bodies of various AppTrack methods so they could observe a) whether or not a method was being executed at runtime and/or b) the values of any variables/parameters used within a method at runtime. This approach to tracing is haphazard at best. Both code scattering and tangling are manifest in the object-oriented implementation of this concern. Scattering occurs because the same piece of tracing code was cut and paste into many different implementation modules. Tangling occurs because numerous classes not designed to cater for tracing behaviour were required to encapsulate tracing logic, leading to the implementation of at least two concerns within one implementation module. Full implementation of the tracing concern in this manner required adding approximately five lines of code to every method of every class. This was a huge undertaking for only limited rewards as tracing is a development concern and this code is removed from production releases of the system.

An aspect-oriented design for this concern would in theory provide pluggable tracing functionality, implemented without affecting any other core concern [12]. This is a stereotypical example of the kind of concern that can be modularised using AspectJ (as illustrated in the next section).

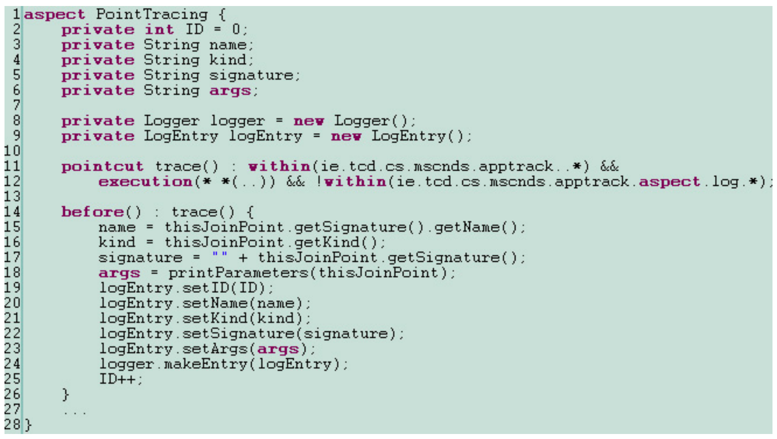
3.1.1 Tracing Aspect Implementation

The implementation of the tracing concern consists of one aspect and two standard Java classes. The `LogEntry` class is a simple bean class. This class contains a variable and a set of accessor methods for each piece of information that is recorded about each method call made during the execution of the system.

The `Logger` class contains the main low-level log-writing functionality used by the logging aspect. `makeEntry(LogEntry)` from the `Logger` class accepts a `LogEntry` object as a parameter, extracts the properties from this object and inserts them into a log file on disk. `makeEntry(LogEntry)` is executed for each method encountered during execution.

The two classes described combine to provide tracing functionality separate from the remainder of the AppTrack codebase. However, they must be associated with the codebase so that they can trace it. For each method that is executed when the system is in operation, a `LogEntry` must be created and the `makeEntry(LogEntry)` method must be invoked. This behaviour is encapsulated within the `PointTracing` aspect, which is shown in figure 1. This aspect associates the tracing concern with the rest of the concerns in the codebase. The `PointTracing` module is an `aspect`, the new implementation module type introduced in AspectJ. This aspect declares a pointcut, `trace()`, line 11, to capture the execution of every method that does not reside within the `ie.tcd.cs.mscnds.apptrack.aspect.log` package. The reason that method calls within the package containing the tracing concern are not logged is to prevent an infinite sequence of logging calls. Once a joinpoint is identified, advice can be associated with it. In this case we chose to run the advice before the execution of each method identified by the pointcut. The aspect contains five class scope variables that correspond to those in the `LogEntry` class. These variables are initialised with the appropriate context specific information, which is obtained from the `thisJoinPoint` variable, lines 15 to 17. The arguments to the currently executing method are returned from the `printParameters(JoinPoint)` method. This method (not shown in figure 1) uses the `thisJoinPoint` variable to generate a suitably formatted string containing the number of arguments, the type of each argument and the actual argument values for the current execution of the method.

All properties of the `LogEntry` class (barring `ID`) are set to the values obtained from the current joinpoint, lines 19 to 23. The information about the current method is then written to the log by passing the `LogEntry` object containing the relevant information to the `makeEntry(LogEntry)` method of the `Logger` class. By encapsulating this behaviour within the `before()` advice component of the `Point-Tracing` aspect we avoid scattering and tangling it throughout each traced method.



```
aspect PointTracing {
    private int ID = 0;
    private String name;
    private String kind;
    private String signature;
    private String args;

    private Logger logger = new Logger();
    private LogEntry logEntry = new LogEntry();

    pointcut trace() : within(ie.tcd.cs.mscnds.apptrack..*) &&
        execution(* *(..)) && !within(ie.tcd.cs.mscnds.apptrack.aspect.log.*);

    before() : trace() {
        name = thisJoinPoint.getSignature().getName();
        kind = thisJoinPoint.getKind();
        signature = "" + thisJoinPoint.getSignature();
        args = printParameters(thisJoinPoint);
        logEntry.setID(ID);
        logEntry.setName(name);
        logEntry.setKind(kind);
        logEntry.setSignature(signature);
        logEntry.setArgs(args);
        logger.makeEntry(logEntry);
        ID++;
    }
    ...
}
```

Fig. 1. Tracing aspect

All code relating to the tracing concern was totally separated from the main AppTrack codebase. Two classes and one aspect were added during the re-implementation and these encapsulate the tracing concern, leaving the remainder of the codebase oblivious[1] to their existence. It is interesting to note here that the tracing concern could have been implemented using an abstract aspect. By leaving the `trace()` pointcut abstract, the `PointTracing` aspect would be totally application independent. The abstract aspect could then be extended by the AppTrack developers, with the pointcut being given an AppTrack specific implementation. This approach is used in the re-implementation of the transactions concern.

3.2 Transactions

AppTrack is a web-based information system. Like most systems of this nature it has an underlying database and can cater for multiple simultaneous users. It is necessary to employ a transaction service in order to retain data integrity while serving the requests of multiple users. Data access is a fundamental functional requirement while a transaction service affecting data access is considered to be a non-functional requirement. This distinction is the cause for their proposed separation in the aspect-oriented re-implementation of the AppTrack system. AppTrack's data access and transactions functionality is implemented together (in the `db` package) resulting in extremely tight coupling. Each key object in the system (e.g., `Applicant, Application`) has a database handler class associated with it to cater for its data access needs (e.g., `ApplicantHandler`). Any class that wants to use data access functionality can only gain access to the classes that will deliver this functionality (handler classes) by going through the `Transaction` class. Consequently, in practice you must declare an instance of the `Transaction` class and obtain an instance of the database handler class you require (e.g., `ApplicantHandler`) from the `Transaction` class. The `Transaction` class initialises the handler object with a connection to the database

[1] "whether the writer of the main code has to be aware that aspects will be applied to it [13]"

and provides implementations of transaction interface methods (commit, rollback etc.) to manipulate this connection as appropriate, meaning that data access and transactions are extremely tightly coupled.

3.2.1 Transactions Aspect Implementation

The first stage in the implementation of the aspect-oriented version of the transactions concern involved deconstructing the existing architecture and removing the dependency relationship between the class providing transactional behaviour and the database handler classes. Following the completion of this task the codebase contained database handler classes for each major object in the system but no facility for obtaining database connections for these handlers to use. The main responsibility of the transactions concern is the establishment, distribution and maintenance of these database connections. The `Transaction` aspect, partially illustrated in figure 2, contains all of the transactional functionality. `Transaction` is an abstract[2] aspect that contains the following methods: `setup(JoinPoint)`, `commit()`, `rollback()`and `freeConnection(java.sql.Connection)`. Alongside these methods are two abstract pointcuts (figure 2, lines 6 and 8): `standardTransaction()` (read-only) and `transactionForUpdate()` (write). The following pieces of advice run at the joinpoints identified by the pointcuts (their behaviour involves calling the transaction methods listed above): `before() : standardTransaction()`, `after() : standardTransaction()`, `before() : transactionForUpdate()`, and `after() : transactionForUpdate()`.

Database handler classes now require a valid database connection object to be passed to their constructor. This connection must come from the class that wants to access the database i.e., the class previously a client of the object-oriented `Transaction` class. The methods in the `Transaction` aspect rely on the existence of a class scope variable of type `java.sql.Connection` named `connection` within any class affected by the transactions concern. It is this connection variable that is manipulated by the transactions concern. All usage of this variable from within the classes making up the transactions concern is achieved via the use of Java's reflection mechanisms. We must note here that AspectJ's introduction mechanism could have been used to introduce the connection variable into the classes requiring data access functionality without actually placing the variable in the classes themselves, further encapsulating the concern implementation. Such a solution would necessitate statically introducing the variable into each relevant class, hence naming each affected class in the `Transaction` aspect. We felt that the loss of genericity associated with this approach was unacceptable.

The `setup(JoinPoint)` method (figure 2, line 10) establishes a connection to the database and sets the `connection` variable in the affected class to have the same value as this new database connection. The `JoinPoint` argument to the method is used to discover the type of the current class, and Java's reflection mechanisms are used to retrieve and manipulate the `connection` variable from the current

[2] The aspect is abstract because we felt that separating the main transactional behaviour from the weaving specification would help reduce overall concern complexity.

```java
abstract aspect Transaction {
    private Connection connection;
    private Field connectionField = null;
    private Class currentClass = null;

    abstract pointcut standardTransaction();

    abstract pointcut transactionForUpdate();

    protected synchronized void setup(JoinPoint jp) {
        try{
            connection = DBConnectionManager.getInstance().getConnection(Constants.POOL);
            connection.setAutoCommit(false);
            currentClass = jp.getThis().getClass();
            connectionField = currentClass.getField("connection");
            connectionField.set(connectionField, connection);
        }
        catch(SQLException sqle) {
            System.out.println("SQLException @ setup: " + sqle.getMessage() + ": in class - " +
                jp.getSourceLocation().getFileName());
        }
        catch(IllegalAccessException iae){
            System.out.println("IllegalAccessException @ setup: " + iae.getMessage() + ": in class - " +
                jp.getSourceLocation().getFileName());
        }
        catch(NoSuchFieldException nsfe){
            System.out.println("NoSuchFieldException @ setup: " + nsfe.getMessage() + ": in class - " +
                jp.getSourceLocation().getFileName());
        }
    }
```

Fig. 2. `Transaction` **aspect - abstract pointcuts and** `setup(JoinPoint)` **method**

```java
public class ApplyForCourseAction {
    public static java.sql.Connection connection;

    public ActionForward perform(...) throws IOException, ServletException
    {
        try {
            CourseHandler courseHandler = new CourseHandler(connection);
            Course course = courseHandler.retrieveByName
                (applyForm.getCourse(), false);
            ...
```

Fig. 3. AppTrack class using the `Transaction` **aspect**

```java
    ...

    before() : standardTransaction() {
        this.setup(thisJoinPoint);
    }

    after() : standardTransaction() {
        this.commit();
    }

    ...
```

Fig. 4. Advice in the `Transaction` **aspect**

```java
aspect WeaveTransaction extends Transaction {

    pointcut transactionForUpdate() :
        execution(ActionForward UpdateDocsListAction.doPerform(..)) ||
        execution(ActionForward UpdateLocationAction.doPerform(..)) ||
        execution(ActionForward UpdateStatusAction.doPerform(..));
```

Fig. 5. `WeaveTransaction` **aspect**

class. The current class can now use its connection variable in declaring an instance of the database handler class it requires. This usage is illustrated in figure 3.

The `commit()` method of the `Transaction` aspect invokes the `java.sql.Connection.commit()` method on the `connection` variable and uses the `freeConnection(Connection)` method to return the connection to the AppTrack connection pool. A call to the `commit()` method makes up the behaviour of the after advice, illustrated in figure 4. The `after : TransactionForUpdate` advice contains some extra behaviour. It first attempts to commit the database operation. If the committal of the operation's effects is unsuccessful then the effects are rolled back.

Still unexplained are the two abstract pointcuts at lines 6 and 8 in figure 2. These pointcuts must be given concrete definitions by the aspect that specifies the crosscutting behaviour i.e., the aspect that extends the abstract `Transaction` aspect.

It is the `WeaveTransaction` aspect that provides a concrete implementation of the `Transaction` aspect, hence implementing the abstract pointcuts. Figure 5 illustrates the concrete implementation of the `transactionForUpdate()` pointcut.

While the aspect-oriented re-implementation of the transactions concern has significantly reduced code tangling, total separation was not possible with the approach we took. The number of transaction-related lines of code evident in each business logic class has been considerably reduced. Each class now contains only one line of code related to the transactions concern. This is the line declaring the `java.sql.Connection` variable that is manipulated by the `Transaction` aspect. Although a positive level of syntactic separation has been achieved, the transactions concern and the core code are still semantically tightly coupled.

3.3 Summary of Aspect-Oriented Re-implementation

At the beginning of this section we listed the full set of concerns that were identified in the original AppTrack codebase and re-implemented using aspect-oriented techniques. Of the seven concerns re-implemented, five were separated completely. However, only two had the property of obliviousness which allows developers to code core business logic without having to take the aspects that affect it into consideration. These two concerns are the development concerns, tracing and enforce factory design pattern. The remaining separated concerns did not exhibit obliviousness because to fully understand the areas of the system to which these concerns are related, developers had to be aware of the existence of the aspects and their effect on the system.

Two concerns could not be fully separated from the AppTrack codebase. The two concerns in question are the most important production concerns that we re-implemented. Both the transactions concern and the exception handling concern could not be entirely separated due to the level of intimacy required with the core codebase that they affect.

4 Impact on Evolvability

A major selling point of the AOSD paradigm is the claim that increased separation of concerns produces the benefit of enhanced evolvability. In this section we assess the evolvability of the AspectJ version of the AppTrack system under the following headings: changeability/extensibility, pluggability, complexity, and lines of code.

As software evolution is intrinsically linked with the act of changing and extending software, the impact on the changeability/extensibility of the software is of great importance. Software system components are often updated or only used in certain contexts. If these components are easily pluggable then system evolution is simplified. For this reason we assess pluggability. The ease with which a system can evolve is directly related to the complexity of the system. If it is logically constructed and can be easily understood by a developer who was not involved in the original development effort, then it is likely that the system can evolve relatively freely. System complexity can also be related to system size, with complexity escalating as codebase size increases. Therefore, we assess the reduction or increase in the number of lines of code required for a concern implementation.

4.1 Changeability/Extensibility

There are two general areas in which the system may change – the aspect itself may need to be modified, or the core system may need to be modified. The AspectJ implementation of the tracing concern can be easily modified to alter the information that is recorded at runtime. This requires making minor, logically related alterations to each of the three classes involved in the implementation. To log a new property about each method executed you would need to add that property to the `LogEntry` bean class, set the property in the `PointTracing` aspect and write the property to the log file in the `Logger` class. To make such a change to the object-oriented implementation would require making changes to every method that is logged, as well as the `LogEntry` and `Logger` classes. The tracing aspect does not need to be considered when altering the behaviour of the core system.

The methods that provide transactional behaviour in the aspect-oriented implementation of the transactions concern are, at a high-level, basically the same as those in the object-oriented implementation. Their actual implementation is made more complex as a result of the use of Java's reflection mechanisms. Hence there is no gain regarding the extensibility of these methods and it could be argued that the heightened complexity hinders their extensibility. However, significant difference is evident in the area of extending the system as a whole. A new class that requires database access now has the option of whether or not this data access should be transactional. The developer is no longer locked into using transactions. Of course, it is generally a good idea to use transactions when accessing data in a distributed environment. To acquire transactional behaviour for a method containing standard database access, a one-line entry must be made in the `WeaveTransaction` aspect naming the method requiring transactional data access. The class that the method resides in must of course contain the `java.sql.Connection` variable necessary for the operation of the transactions concern. The increased encapsulation of the transactions concern makes general extension of the AppTrack system less complex and time consuming, as programming for transactions is no longer a major coding issue. However, the increased complexity of the solution as a whole means that changing/extending this area of the system requires a greater knowledge of the implementation of the concern than before. This is discussed in greater detail in section 4.3

4.2 Pluggability

The tracing concern is completely pluggable. The three classes composing the concern implementation do not define any dependencies with any classes outside of the package in which they reside. This means that a production release of the code can be generated simply by excluding the tracing package when compiling/weaving the system. This is far more convenient than manually deleting scattered and tangled tracing code from every affected method across the codebase.

The transactions concern is not cleanly pluggable in the same manner as the tracing concern. The solution is designed to work with a database connection declared in each class requiring transactional data access. The `Transaction` aspect is responsible for initialising, manipulating and closing connections to the database. Unplugging the aspect and recompiling the system would result in a total loss of database connectivity due to connections not being initialised. However, the amount of effort required to gain standard data access capabilities (should they ever be required) following the removal of the aspect-oriented implementation is minimal when compared to removing the use of transactions in the object-oriented version and using standard non-transactional data access.

4.3 Complexity

The integration of the aspect-oriented tracing concern with the AppTrack codebase reduces overall system complexity. Although there is a learning curve involved to get up to a sufficient level of understanding and competence with AspectJ, the modularity achieved by the new implementation makes the remainder of the codebase more streamlined, readable and generally more comprehensible. The re-implementation also enforces consistent behaviour. All methods are treated equally i.e., the same information is logged for each method. This was not necessarily the case with the object-oriented solution (due to mistakes inherent to scattering), a phenomenon that often caused confusion. It is our view that the benefits of the new solution negate the disadvantage of the learning curve involved in understanding the solution.

Despite the reduction in tangling and increased separation of the transactions concern, we consider the aspect-oriented implementation of the concern to be more complex than its object-oriented counterpart. This complexity arises for a number of reasons. Firstly, the use of Java's reflection mechanisms to create generic versions of methods that can cater for all classes affected by the concern without explicitly naming them complicates the implementation. The methods of the `Transaction` aspect require significant study before a comprehensive understanding is attained. Secondly, each class that contains methods requiring transactional data access must declare a variable of type `java.sql.Connection`. For an observer of the code looking at a class containing data access code, this can be quite confusing. In addition, the base system appears to work without the initialisation or committal of the database connection held by the class. The data access portions of the system can only be understood with full knowledge of the relatively complex, reflection-based, transactions concern. According to [14], due to the nature of the concern, separation of transaction interface methods is syntactic rather than semantic. This, they argue, is because transactions should be implemented with the rest of the application semantics. They state that AOSD and specifically the AspectJ programming language can be used to achieve some level of syntactical separation, but that the developer should be aware of its very syntactic-only nature. It appears that this is the case with the re-implementation of the transactions concern in the AppTrack system. This adds to the complexity of the new implementation.

4.4 Lines of Code

Approximately 360 methods are logged by the tracing concern, the implementation of which accounts for 105 lines of code in total. If the object-oriented tracing concern was implemented across the whole codebase then each method would require augmentation with approximately 5 lines of tracing code. As well as this, the two supporting classes required, `LogEntry` and `Logger`, make up 60 lines of code. This brings the estimated total number of lines of code needed for a full object-oriented implementation to 1860. The aspect-oriented solution requires only 5.65% of the code necessary for the object-oriented solution.

One whole class (the object-oriented `Transaction` class) and 33 lines of tangled transactions code were removed from the AppTrack codebase as a result of the aspect-oriented re-implementation. However, the aspect-oriented implementation of the module providing the transactional behaviour is 16 lines longer than the object-oriented version it replaces. This means that a total saving of 17 lines of code was achieved with the aspect-oriented implementation of the transactions concern.

5 Related Work

The authors of [15] describe the implementation of a web-based information system using an early version of AspectJ. They outline four aspect/class associations and state that they employed a policy of only using class-directional aspects, where the aspect knows about the class(es) it affects but the opposite is not true. The tracing aspect adheres to this association, whereas the classes affected by the transactions aspect rely on its existence. We have seen the disadvantages of this relationship in the implementation of the transactions concern. Overall, the authors concluded that the use of AOSD techniques in the development of a web-based system resulted in a fast, well-structured system in a reasonable amount of time. The authors of this paper do not address evolvability explicitly, which is the focus of this paper.

Although not presented in this paper, AppTrack's exception handling concern was re-implemented with AspectJ. In [16], Lippert and Lopes describe their investigation into the ability of AOSD techniques to ease the tangling and scattering related to exception handling in standard Java applications. They conclude that the use of AOP, specifically AspectJ, can drastically reduce the portion of application code related to exception detection and handling. More significantly, along with the reduction in code size, they found that AspectJ provides better support for different configurations of exception handling behaviour with respect to standard Java. Greater support for reuse, incremental development, cleaner program texts and automatic enforcement of contracts were also achieved. While our implementation of AppTrack's exception handling concern was different to their implementation (due to structure imposed by the Struts framework), we believe there is consistency between the two sets of findings.

A number of experiments were carried out at the University of British Columbia to assess the capabilities of AOP in two main areas – debugging and change. The results of these experiments, which involved participants undertaking programming exercises using AspectJ and a control language, are described in [17]. The results of the experiments led the researchers to two key insights. The first of these is that programmers may be better able to understand an aspect-oriented program when the effect of the aspect code has a well-defined scope. Our findings support this conclusion. The tracing concern, with a well-defined scope, is relatively easy to understand. In contrast, the scope of the complex transactions concern is less well-defined. This concern affects various methods that are spread throughout classes contained in numerous different application packages. The second insight is that the presence of aspect code may alter the strategies programmers use to address tasks perceived to be associated with aspect code. We did not conduct any programmer performance comparison analysis. The researchers behind this study did not use concerns similar to ours in their experiments and hence do not discuss implementation issues akin to those experienced during the re-implementation of the transactions concern.

The area of aspect-oriented implementations of transactions concerns has been the subject of two papers in recent times. The authors of [14] begin their paper by highlighting that they feel it does not make sense to use AOP techniques to separate concurrency control from the other parts of a distributed system. They state that while AOP, specifically AspectJ, can be used to achieve some level of transaction-related code separation, the separation achieved is purely syntactic rather than semantic. Their attempts at analysing the extent to which it is possible to aspectise transaction interfaces, like ours, resulted in separation that is, in some cases, artificial and leads to rather confusing code. The authors of [18] argue that the kind of transparency sought by the authors of [14] should not be confused with obliviousness, which is supported by AspectJ and allows an application programmer to not worry about inserting hooks in the code so it is later affected by the aspects. They state that this does not mean that the programmer should not be aware that the aspects intercept the application code. Likewise, the programmer should be aware of the code that their aspects affect. They state that in this sense there may be strong dependencies between AspectJ modules and standard classes, reducing some of the benefits of modularity, including evolvability. It is clear from our approach to separating AppTrack's transactions concern with AspectJ that hooks (in the form of the specifically named database connection variable) did have to be added to the application code to cater for the behaviour of the aspect that affects them. We feel that this gives more weight to the findings of Kienzle and Guerraoui in [14].

Finally, Alexander and Bieman discuss the challenges of aspect-oriented technology in [20]. The paper seeks to understand both the strengths and weaknesses of AOSD and to raise awareness of the potential negative side-effects of its use. The authors seek to assess whether or not the benefits created by AOSD are worth the potential negative side-effects. The paper does not put forward a concrete stance on this issue.

6 Conclusions

We discussed our experience re-implementing an object-oriented web-based information system with AspectJ, a programming language that supports the aspect-oriented software development paradigm. We then considered the effects this re-implementation process had on the evolvability properties of the system. In the new version of the system, code relating to the implementation of tracing and transactions concerns is separated from the core application codebase as much as was possible with the approach we took. This separation is total in the case of the tracing concern and partial in the case of the transactions concern.

Our main contribution is to evaluate the effect that the use of the AOSD techniques supported by AspectJ has on the evolvability of computer software. While the effect on evolvability brought about by the re-implementation of the transactions concern is considered less than impressive, the evolvability of the tracing concern is greatly improved. The ease of maintenance afforded by the aspect-oriented implementation is a source of heightened productivity. Changeability, pluggability and comprehensibility, important properties in relation to maintenance, are greatly enhanced in the new version of the tracing concern. Statements made in [19] support our conclusions regarding maintenance and productivity. The author asserts that not only is it true that the real benefits of AOSD are seen in the latter stages of the software development lifecycle, but that this behaviour is vital. Without the benefit of simplified maintenance and increased productivity, AOSD would fail to deliver on some of its key promises. This paper shows that for certain concerns, implemented with AspectJ, it does not fail to deliver. However, the heightened complexity of the transactions concern can actually serve as a hindrance to maintenance productivity levels if the developer undertaking the maintenance task is not familiar with both the AppTrack application and AOSD (and the relevant AOP mechanism, in this case AspectJ).

Our findings lead us to the conclusion that the extent to which evolution is affected is directly related to the type of concern that is implemented. The implementation of the tracing concern has a well-defined scope. It is confined to the classes that compose the tracing package and those classes define no restrictive dependency relationships. The concern's behaviour has global scope in that it affects the entire application, hence it does not deal with any application specific components. For these reasons total separation was possible and the evolvability benefits are substantial. Conversely, the implementation of the transactions concern affects only specific points in the application and defines restrictive dependency relationships with the core code that it affects. For these reasons the affect on evolvability is less positive. Having that said, given a sufficient knowledge of the concern, the ease with which it can be modified is enhanced (as described in section 4.1).

The conclusions drawn from our evaluation of the re-implemented AppTrack system indicate that there is a definite value associated with the practice of re-implementing an existing object-oriented codebase using AOSD techniques as supported by AspectJ. The separation afforded by these techniques leads to benefits in the evolution stage of the software lifecycle. However, these benefits may not be offset by other issues that arise as a result of the separation. We conclude that the value added to system evolvability afforded by an AspectJ re-implementation can indeed be significant. However, it is directly related to the genre of the concern that is re-implemented.

Acknowledgements

Many thanks to Elisa Baniassad and the anonymous reviewers for their comments on early drafts of this paper.

References

1. E W. Dijkstra. "*A Discipline of Programming*". Prentice-Hall, 1976.
2. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, J. Irwin. "*Aspect-Oriented Programming*". ECOOP, 1997.
3. B. Meyer. "*Object-Oriented Software Construction*". Prentice-Hall, 1997.
4. S. Clarke, W. Harrison, H. Ossher, P. Tarr. "*Subject-Oriented Design: Towards Improved Alignment of Requirements, Design and Code*". OOPSLA, 1999.
5. http://www.aosd.net – The aspect-oriented software development website: December 2nd, 2002.
6. http://jakarta.apache.org/struts/ – The Struts page on the Apache Jakarta project website: December 2nd, 2002.
7. E. Gamma, R. Helm, R. Johnson, J. Vlissides. "*Design Patterns. Elements of Reusable Object-Oriented Software*". Addison Wesley, 1995.
8. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold. "*An Overview of AspectJ*". ECOOP 2001.
9. R. Bodkin. "*Aspect-Oriented Programming with AspectJ*". Slides from a talk given at SDWest, 2002.
10. E. Hilsdale, J. Hugunin. "*Introduction to Aspect-Oriented Programming with AspectJ*". Tutorial 3, AOSD 2002.
11. R. Jeffries, A. Anderson, C. Hendrickson, K. Beck. "*Extreme Programming Installed*". Addison-Wesley, 2000.
12. The AspectJ Team. "*The AspectJ Programming Guide*". Available from the official AspectJ website – http://www.aspectj.org.
13. T. Elrad, R. Filman, A. Bader. "*Aspect-Oriented Programming*". Communications of the ACM. October 2001 - Volume 44, Number 10. p31.
14. J. Kienzle, R. Guerraoui. "*AOP: Does it Make Sense? The Case of Concurrency and Failures*". ECOOP, 2002.
15. M. Kirsten, G. Murphy. "*Atlas: A Case Study in Building a Web-Based Learning Environment using Aspect-Oriented Programming*". OOPSLA 1999.
16. M. Lippert, C. Lopes. "*A Study on Exception Detection and Handling Using Aspect-Oriented Programming*". ICSE 2000.
17. R. Walker, E. Baniassad, G. Murphy. "*An Initial Assessment of Aspect-Oriented Programming*". ICSE 1999.
18. S. Soares, E. Laureano, P. Borba. "*Implementing Distribution and Persistence Aspects with AspectJ*". OOPSLA, 2002.
19. J. Memmert. "*AOP and Evidence of Improvements*". Thread on the aosd-discuss mailing list. February 20th, 2002.
20. R. Alexander, J. Bieman. "*Challenges of Aspect-Oriented Technology*". ICSE, 2002.