* Presented at the 1998 International Workshop on Aspect-Oriented Programming (Kyoto, Japan; April 20, 1998) held at the 20th International Conference on Software Engineering.

Assessing Aspect-Oriented Programming and Design: Preliminary Results

Robert J. Walker, Elisa L. A. Baniassad, Gail C. Murphy
{walker, bani, murphy}@cs.ubc.ca

Department of Computer Science
University of British Columbia
Vancouver, B.C. V6T 1Z4 Canada

Technical Report TR-98-03

Abstract

Aspect-oriented programming is a new software design and implementation technique proposed by researchers at Xerox PARC. This project is assessing the claims of aspect-oriented programming to improve the software development cycle for particular kinds of applications. The project is divided into three experiments, the first of which has been completed. These experiments have been designed to investigate, separately, such characteristics of aspect-oriented development as the creation of new aspect-oriented programs and ease of debugging aspect-oriented programs.

1 Introduction

Two of the most important and most difficult questions one can ask about a new design or programming approach are whether the approach is useful and whether the approach is usable. One way to evaluate these questions is to make the design or programming approach immediately accessible to the greater community and to simply see whether the approach sinks or swims. Although ultimately the goal is to positively affect the greater community through the adoption of the approach, this strategy has many pitfalls: useful techniques that are not quite usable can be lost, and usable techniques that are not particularly useful can mask the adoption of other, perhaps more powerful, techniques. Aspect-oriented programming is a new software design and implementation technique proposed by researchers at Xerox PARC[2].

This technique is in its infancy. The aspect-oriented approach claims to make it easier to reason about, develop and maintain certain kinds of application code while maintaining highly efficient code. To better understand the usefulness and usability of the aspect-oriented approach, we are currently conducting three experiments. These experiments are designed to investigate such characteristics of aspect-oriented development as the creation and ease of debugging programs built with aspect-oriented design and programming. The experiments investigate aspect-oriented design and programming as represented in AspectJ†, an aspect-oriented variant of Java‡ developed at Xerox PARC.

In conducting software engineering experiments, we are constrained by four factors: *the pool of potential participants* available to us is very small, *the amount of time* each participant can devote to an experiment is very short, especially in comparison to typical development times of even tiny applications, *the cost of running and analyzing experiments is high*, and since the evaluation of an aspect-oriented approach is complex, *some precision of measurement needs to be forfeited* in favour of realism[3]. As a result, our "experiments" were set up as semi-controlled empirical studies rather than statistically valid experiments.

This report describes the results of a pilot study and the first experiment. The pre-study was used to test our experiment design in the context of investigating whether aspect-oriented programming eases the creation of correct program. The first experiment studied the ease of debugging such a program. The two experiments which have yet to begin are designed to investigate the ease of change of an existing program and the ease of design and implementation of a new program.

2 Pilot Study

Before designing our set of experiments, it was necessary to understand how difficult a problem we could realistically ask a participant to tackle in a period of no more than four hours.

† AspectJ is a trademark of Xerox Corporation.
‡ Java is a registered trademark of Sun Microsystems.

We also used this pilot study to determine better methods of training and questioning our participants.

The experimental question approached in the pilot study was whether, in the context of AspectJ, the combination of JCore for component programming and COOL as a synchronization aspect language eases the creation of multi-threaded programs compared to programming in the Java object-oriented language.

The basis of this experiment was to select a small programming problem with concurrency, and then have several Java-knowledgeable programmers attempt to produce a solution to the problem, some working in Java, others in AspectJ. Since we were working within a small community from which to draw participants we ranked the participants and chose the least qualified for this pilot study. We held the following questions in mind while watching and annotating the video-tapes of the sessions:

- Can programmers working with an aspect-oriented language produce a "correct" program in less time than programmers working with an object-oriented language?[§]

- Do programs produced with an aspect-oriented programming language have fewer bugs than programs created with an object-oriented programming language?

2.1 Format

A pilot study session proceeded in stages. First, we required that the experimental participant review materials on concurrent programming. If the participant was to use AspectJ, the experimenter would introduce the concepts of aspect-oriented programming and provide the participant with an opportunity to become familiar with AspectJ; otherwise, they were given material describing Java synchronization usage [1]. Participants using AspectJ were required to examine an example piece of code to ensure that they were familiar with the syntax of the language. Finally, the participants were told the problem, provided a programming environment, were given 1.5 hours to complete the assigned task, and were asked to think aloud while we video-taped their progress. Twice during each session, the invigilator asked the participant a set of questions.

The programming problem chosen for the experiment was a simple version of a non-audio karaoke machine in which text at the bottom of a small window scrolls from right to left and a ball bounces straight up and down above the text. The problem was to synchronize the ball and the text such that the ball would bounce on the start of each word. The text or the ball could be paused to permit this to happen. The participants were provided with a skeleton program from which to begin that contained the basic functionality to make the ball bounce and the text scroll.

Six participants took part in the experiment, three used Java to program their solution and three were given AspectJ.

2.2 Results

None of the six participants in the experiment were able to produce a solution to the programming problem in the time provided, although two came close (one Java, one AspectJ). We analyzed this pilot study to learn how to conduct subsequent studies; for this reason we did not go to great lengths to guarantee the abilities of our participants. In addition, the difficulty of the task was greater than we had expected: aside from the basic point of encoding the necessary synchronization, the means for completing the non-synchronized aspects of the program, the semantics of the problem domain, and the functionality of the existing code (written by someone else) needed to be understood first. One and a half hours was unreasonably short.

The following lessons are among those we were able to take away from this pilot study:

- Participant training with and set-up of the programming environment is necessary. It is impossible to test usefulness when usability is at a minimum. To address this, we modified our experiment design by ensuring that all participants were given a thorough lesson on how to use the environment.

- All participants must be given lengthy exposure to the synchronization mechanisms provided by the language they are given to use. After this exposure they should be tested to ensure they know the information necessary to perform the experiment task. Subtle differences in synchronization constructs can be a great hindrance to someone attempting to use a language in which they have not frequently programmed synchronization, even if they are otherwise familiar with synchronization concepts. In the first experiment we ensured that all participants spent the same amount of time reading the synchronization documentation, and asked them to describe the basic concepts necessary for completion of the experiment task.

An interesting observation was that, among the participants, only the two near-successful ones pursued a course of action expressing a separation of concerns: get the code to work without synchronization first, then add the synchronization.

2.3 Participants' Comments

In general the participants that used AspectJ liked the method even though they were unable to complete the problem. We asked each of the AspectJ participants if they had any comments about aspect-oriented programming in general.

[§] Correct here is used to mean a program that meets the specification given for the program.

> "Well, [it was] just the way I imagined that [aspect-oriented programming] would be used in a specific program. I thought it was really cool, because I could concentrate on what I was doing now, on the functionality that it would have by itself, as opposed to how to synch it up with the other object."

> "Ultimately, I guess the idea is that the objects could be separated so you could change how the coordination was done without messing with the objects. I always like that, changing one little thing without touching what's going on in the other place. It has a really elegant nature to it."

2.4 Pilot Study Critique

Problems involving concurrency are hard to solve. One interpretation of the results of this experiment is that the support for concurrency in AspectJ did not ease the difficulty of the programming problem sufficiently such that it could be solved with the aspect-oriented approach when it could not be solved with the object-oriented (Java) approach. This result is not surprising in that aspect-oriented programming is meant to ease the expression of the solution to the problem rather than to necessarily help the software engineer design the solution. From the participants comments, it appears that many of them had difficulty framing an appropriate solution in the time available.

The participant who made the most progress on the problem was a participant using the aspect-oriented approach. Only a few small changes were necessary to the coordinator code produced by this participant to produce a solution to the problem: the changes were all of the same nature — an attempt to coordinate on objects rather than classes.

Running Time

The participants were given 1.5 hours to program a solution to the given problem. Given that no participant was able to solve the problem, it is clear that either the problem was too complex, or insufficient time was provided to the participants. We had thought that 1.5 hours to solve the problem from a given code base would be reasonable given that the initial solution was coded, from scratch, in just over 2 hours; however, this did not include the many hours spent discussing the semantics of the karaoke machine synchronization during the design of the pilot study.

This problem could be mitigated in two ways. First, additional dry-runs could be held to try and gauge if the experimental running time is reasonable. Second, the experimental procedure could have called for giving participants as much time as they needed up to some (reasonably) large maximum such as 3 hours. However, given that only two of the six participants were pursuing approaches likely to be successful, it is unclear that additional time alone would necessarily lead to more consistent (and interpretable) results. Alternatively, a more standard problem based on readers and writers might have been chosen. One advantage of choosing a non-standard problem like karaoke was that the participants could not simply provide a 'textbook' solution.

Code Skeleton

To focus the participants' efforts on concurrency we provided them with a code skeleton from which to start programming. This code skeleton lacked any synchronization and also lacked some of the functionality necessary so as to allow the participant some flexibility in pursuing a solution. In retrospect it is clear we did not provide a sufficient overview of the existing code or sufficient time for the participant to review the code and ask questions. A specific period of time for review and questions might have mitigated problems arising from a lack of understanding of the given code skeleton.

Participant Selection

Since this was seen as a pilot study and we had a limited number of potential participants we put the programmers with less experience in this study. All participants were asked prior to selection whether they were familiar with the concepts of concurrency and Java; however, since the aim of this pilot study was mainly to gain experiment design information, stress was not placed on the screening of participants. We relied on interviewing and questioning of the participants on these topics rather than on a specific pre-test.

More than one participant also spent a significant amount of the experiment time trying to understand syntax errors, both from Java and AspectJ. The latter have since been clarified via improvements in the AspectJ weaver.

3 Experiment 1: Ease of Debugging

The intent of this experiment was to learn whether the separation of concerns provided in aspect-oriented programming enhanced users ability to find and fix functionality errors (bugs) present in a multi-threaded program. In terms of AspectJ, the question was whether the combination of JCore for the component programming and COOL as a synchronization aspect language eased the debugging of multi-threaded programs, compared to the ability to debug the same program written in Java.

A 600 line, multi-threaded, program was created, and three synchronization bugs were introduced. Then, pairs of programmers, knowledgeable in multi-threaded programming techniques and object-oriented programming, attempted to fix the three bugs. Three of the pairs worked with AspectJ, three with Java. The solutions to the program were compared in the following ways:

- Can programmers working with an aspect-oriented programming language debug a multi-threaded program in less time than programmers working in an object-oriented language?
- Are programmers debugging an aspect-oriented programming language able to more quickly and easily identify the cause of a bug in a multi-threaded program than in one written in an object-oriented language?

3.1 Format

The program provided to the participants was a simple digital library consisting of 6 classes, 3 of which required coordination. The library had two main actors: readers and libraries. The readers would make requests to libraries for a particular book. Libraries would search within their internal repositories for the book, and also ask remote libraries to do the same. Each reader could query one library, and each library could directly query at least one other. Three synchronization bugs were inserted into the code.

The participants worked in pairs[¶]. In each pair, one participant had control of the computer with the programming problem, and the other had access to a report describing the symptoms of the bugs, and on-line documentation. They were then asked to fix each bug in turn,

The bugs were cascading, meaning that the symptoms of the first hid the symptoms of the second, and the second hid those of the third. In the first bug only one reader would make a request and then they system would halt. The participants had to remove a per-class self-exclusive coordination on the `run()` method of the `Reader` class so that more than one reader could run. In the second problem, two readers would make requests and then the system would deadlock. The participants were required to determine that the deadlock occurred when two libraries each tried to do a remote-search on the other at the same time. They then had to remove a per-object self-exclusive coordination on the `remoteSearch()` method of the `Library` class so that the system would no longer deadlock. The third bug was that more than one reader was able to check out the same book from the same library. For this problem, the participants had to add a per-object self-exclusive coordination on the `checkOut()` method of the `Library` class so that only one reader could check out a book at a particular library at a time.

To compare Java with AspectJ, a pair of synchronization lock classes were built which were identical in functionality with the woven output from AspectJ source code. This allowed the true aspect-oriented properties of COOL, as opposed to its library-like functionality, to be compared with non-aspect-oriented Java code.

¶Participants were graduate students in computer science, and an undergraduate in computer engineering.

The experiment consisted of six pairs of participants, three worked with Java and the others with AspectJ. All of the pairs were given time to train to familiarize themselves with the languages they were to be using; 1.5 hours were allowed each pair to code their solution. Each of the pairs were to be asked for reports of their progress either after they had coded each of the solutions, or at 1/2 hour intervals, whichever came first.

3.2 Results

In each the AspectJ and Java groups, all of the pairs of participants were able to find and correct all three of the bugs. We examined the performance of the pairs by comparing the time it took them to fix each of the bugs, how many times they built and ran the program, how many times they examined the semantics of the core functionality of the program, if they mixed synchronization and core functionality issues, if they searched for a synchronization solution by modifying the core functionality, and also the number of times the pairs changed the file they were examining while reaching their solution. We first discuss each data element in isolation, and then correlate and summarize the results.

Time

The completion times for each of the three bugs are shown in Figure 1. The largest difference in completion times was with respect to the first bug; the AspectJ teams clearly repaired the bug faster than the Java ones. For the second and third bugs, there was a smaller difference. When examining the time information in isolation we are unable to draw any definite conclusions. The quicker AspectJ time in the first bug could be attributed to any number of factors, and could imply that COOL is an easier language to quickly understand, or that the bug was more obvious when using COOL than Java. In the data correlation section the distribution of completion times is discussed with relation to the amount of programming understanding necessary to complete the programming task.

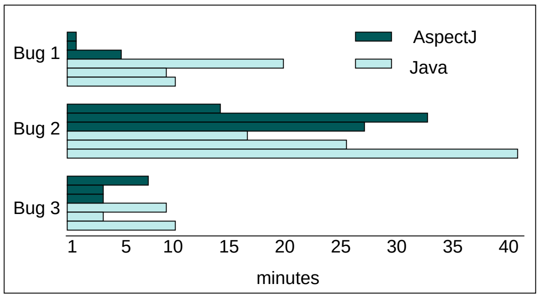

Figure 1: Completion times

Switching Between Files

We were interested in determining if, for bugs where more semantic analysis was being performed on the code, users had to switch between files more using AspectJ because of the need for context of the synchronization code. For this reason, we recorded the number of times the pairs switched the file they were examining. Figure 2 shows that the AspectJ pairs made fewer file switches than the Java group for bug 1, more for bug 2 and slightly less for bug 3.

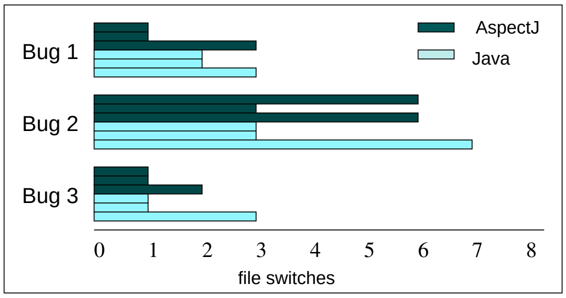

Figure 2: Number of file switches

Instances of Semantic Analysis

The histograms shown in Figure 3 highlight the difference in number of instances of semantic analysis over the nine sessions. To determine the number of instances of semantic analysis we recorded the number of times participants said something to the effect of "let's find out what this does...". This indicates that the Java pairs spent more time analyzing the actual behaviour of the code than the AspectJ pairs did. In the AspectJ session with the most instances of semantic analysis, the group members openly disagreed as to how much semantic analysis was necessary to solve the second bug:

> A: ...we *know* it's in the COOL file...
> B: But we have to know what they *do* before changing anything.

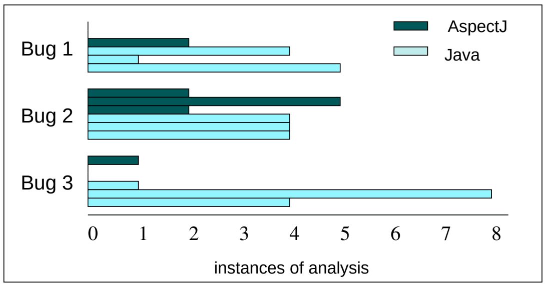

Figure 3: Instances of semantic analysis

Builds

For the first and third bugs, there was only one build per bug with the exception of one Java pair for the first bug, who built and executed five times, and one Java pair in the third bug, who built and executed twice. There was no direct correlation between builds and instances of semantic analysis; however, there was a slight correlation between the number of builds performed and the number of file switches.

Mixing Concurrency and Functionality Issues

In each of the AspectJ and Java groups, one group attempted to solve the synchronization bug with a change to the core functionality of the code.

Granularity Analysis

Since AspectJ synchronization is fixed at a method-level granularity, users of Java have an opportunity to think about the granularity of locks that AspectJ users do not. To collect the instances of this we noted when the users attempted to move locks around within a method, hence implementing a finer granularity of locking than the original method granularity. Only one Java pair investigated locking granularity in the first bug, one in the second, and two in the third. None of the AspectJ participants questioned the synchronization granularity imposed by COOL.

Correlation of Data

When examined in isolation, the increase in number of file switches made by the AspectJ pairs versus the less significant increase for the Java pairs from the first to the second bug may be explained by the fact that the Java groups had done extensive initial file investigation in solving the first bug.

However, the AspectJ group spent less time performing semantic analysis than the Java group did. This could explain why the times for the Java pairs never caught up to those of the AspectJ group. The Java group's general lack of regard for the granularity of locks removes this as an explanation for the extra time spent. One other point must be clarified regarding time: Both the AspectJ and Java pairs spent relatively equal time in building and executing their program. The additional time for weaving AspectJ was negligible.

The number of instances of semantic analysis somewhat explain the number of file switches made by the AspectJ pairs. In the second bug (the bug with the highest average of semantic analysis instances) the most file switching occurred. We believe that there were less file switches by AspectJ pairs than Java pairs on bug 2 because less semantic analysis was performed to solve the bug.

Participants' Thoughts

We asked the AspectJ users what they thought about the separation of the synchronization code from the rest of the core. Two of the three groups were enthusiastic and noted that they did not want the code for the coordination in-line:

> I'd much rather have it separated like this. I really would. ... I would rather not look at the details

> It meant that since [the problems] were just synchronization problems we just had to look at the parts that were related to synchronization. We could have spent lots of time looking at the non-synchronization parts, at one point we did look briefly, but it was clear there was nothing about synchronization in that code, and the only way to deal with synchronization was to look in the COOL files.

The other group felt that COOL provided a handy way of summarizing coordination of and between methods, but were unhappy with the actual separating out of the coordination code.

> The only place I can see there could be an advantage is if you know that you have some modules you are working with that are tested and you are *sure* you can limit the bugs to synchronization issues in which case you don't really have to understand the code.

They would have opted instead for the COOL code to have been inserted in pertinent places throughout the code so that the user could see in once glance both the coordination and the method at the same time. Interestingly, this pair (the third AspectJ pair) switched less between files in total than any of the Java pairs.

We asked the Java groups how they mentally separated the synchronization code from the core code. One participant noted that when looking at Java synchronization code they made no algorithmic differentiation between the synchronization code and the core code. They continued by discussing the need for some abstraction of the synchronization that was higher level than the locking available through Java. "Some way of specifying that you have certain constraints between methods within classes or objects, instead of using this scheme". They noted that this shortcut for locking would save both five lines of code and save you looking at the code itself. We admit that this is speculative since the Java people had no experience with the real separation.

4 Summary

With the first of our three experiments, we were able to obtain interesting indications about the use of aspect-oriented programming versus object-oriented programming. We noted that users of the aspect-oriented programming language AspectJ were able to complete debugging tasks with fewer instances of semantic analysis which seemed to lead directly to less switching between files, indirectly to fewer builds, and ultimately to quicker completion times.

We used a pilot study to gather a set of guidelines about the design of further studies, and used those guidelines in designing the first main experiment. The ability of one participant to come close to an appropriate solution with a coordinator demonstrates it is possible to learn the approach quickly and apply it.

The first experiment highlighted the usefulness of being able to easily express and understand synchronization code. We learned that there are times at which it is useful for synchronization code to be embedded in the core functionality, but that at times work can be speeded considerably (as in bug 1) when synchronization code is separated from the rest.

5 Acknowledgments

We would like to thank the Xerox Embedded Computation Area group for their comments on the experiment concepts and the use of the AspectJ weaver, the anonymous participants who took part in the sessions, and Robert Rekrutiak and Paul Nalos for their work on experiment setup.

Funding provided by Xerox Corporation and a UBC Graduate Fellowship.

References

[1] K. Arnold and J. Gosling. *The Java™ Programming Language*. Addison-Wesley, 1996.

[2] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, and J. Irwin. Aspect-oriented programming. In *ECOOP '97 - Object-Oriented Programming. 11th European Conference Proceedings. Jyvaskyla, Finland*, pages 220–242, June 1997.

[3] Joseph E. McGrath. Methodology matters: Doing research in the behavioral and social sciences. In Ronald M. Baecker, Jonathan S. Grudin, William A. S. Buxton, and Saul Greenberg, editors, *Readings in Human-Computer Interaction: Toward the Year 2000*, pages 152–169. Morgan Kaufmann, San Francisco, 2nd edition, 1995.