An Initial Assessment of Aspect-oriented Programming

Robert J. Walker, Elisa L.A. Baniassad and Gail C. Murphy
Dept. of Computer Science
University of British Columbia
201-2366 Main Mall
Vancouver, BC V6T 1Z4 Canada
{walker,bani,murphy}@cs.ubc.ca

Technical Report TR-98-12 (Revision of TR-98-03)

ABSTRACT
The principle of separation of concerns has long been used by software engineers to manage the complexity of software system development. Programming languages help software engineers explicitly maintain the separation of some concerns in code. As another step towards increasing the scope of concerns that can be captured cleanly within the code, Kiczales and colleagues have introduced aspect-oriented programming. In aspect-oriented programming, explicit language support is provided to help modularize design decisions that cross-cut a functionally-decomposed program. Aspect-oriented programming is intended to make it easier to reason about, develop, and maintain certain kinds of application code. To investigate these claims, we conducted two exploratory experiments that considered the impact of aspect-oriented programming, as found in AspectJ version 0.1, on two common programming activities: debugging and change. Our experimental results provide insights into the usefulness and usability of aspect-oriented programming. Our results also raise questions about the characteristics of the interface between aspects and functionally-decomposed core code that are necessary to accrue programming benefits. Most notably, the separation provided by aspect-oriented programming seems most helpful when the interface is narrow (i.e., the separation is more complete); partial separation does not necessarily provide partial benefit.

Keywords
qualitative assessment, separation of concerns, empirical study, software design, debugging, source code evolution

1 INTRODUCTION

The principle of separation of concerns [7] has long been used by software engineers to manage the complexity of software system development. Many programming languages provide explicit support for separation of concerns by providing different sub-languages for expressing the structure of data versus the functionality to be performed on the data. Pascal [11] is one example of a language with this design. Software specifiers and designers also use the principle when using notations, such as UML [5], which place structure and function information into separate diagrams.

Aspect-oriented programming is a new programming technique that takes another step towards increasing the kinds of design concerns that can be captured cleanly within source code [12]. Aspect-oriented programming provides explicit language support for modularizing design decisions that *cross-cut* a functionally-decomposed program. Instead of spreading the code related to a design decision throughout a program's source, a developer is able to express the decision within a separate, coherent piece of code. For example, ensuring a set of operations do not concurrently execute typically requires spreading code throughout the operations; an aspect-oriented approach allows the synchronization constraint to be specified in one separate piece of code. The aspect code is combined with the primary program code by an aspect weaver. Several different aspect-oriented programming systems have been built, including AML [10], an environment for sparse matrix computation, and RG [17], an environment for creating image processing systems.

The aspect-oriented approach claims to make it easier to reason about, develop, and maintain certain kinds of application code [12]. To begin assessing these claims, we undertook a series of exploratory qualitative studies, including both case studies and experiments[1] [18]. The case study format allowed us to investigate broad usefulness and usability questions surrounding the approach. The experiment format allowed us to focus on more specific questions related to the claims of the technique.

This paper reports on two of the exploratory experiments we conducted to investigate aspect-oriented programming. A particular aspect-oriented programming language created by researchers at Xerox PARC, called AspectJ™ (version 0.1) [1], was used in these studies. This version of AspectJ uses a slightly modified form of Java™ [8] for express-

[1] We use the term experiment similar to Basili: "a study undertaken in which the researcher has control over some of the conditions in which the study takes place and control over (some aspects of) the independent variables being studied" [2, p. 444].

ing the core functionality of a program, and supports two aspect languages: `Cool` for expressing synchronization concerns, and `Ridl` for expressing distribution concerns.[2]

Each of the two experiments considered a different programming activity. In the first experiment, we considered whether the separation of concerns provided by AspectJ enhanced a developer's ability to find and fix faults present in a multithreaded program. The second experiment focused on the ease of changing an existing distributed system. In each case, we compared the performance and experience of programmers working in AspectJ with those of programmers working in a control language: Java in the case of the debugging experiment, and Emerald [3] in the case of the change experiment.

The results of these experiments highlight the importance of the *aspect-core interface* in achieving development benefits with aspect-oriented programming. The aspect-core interface refers to the boundary between code expressed as an aspect and the functionally-decomposed code. This interface is *narrow* when the scope of the effect of an aspect across the boundary is well-defined, and when the aspect can be reasoned about without extensive analysis of the core code. In the experiments, the narrow interface provided by the synchronization aspect language helped the participants to complete assigned tasks. In contrast, the wider interface provided by the distribution concern language seemed to hinder participants.

Our experiments also indicate that aspect-oriented programming may alter the programming strategies used by developers. Specifically, programmers may be more likely to first try to solve a problem related to a concern captured as an aspect by initially focusing on the aspect code. This new strategy tends to help when the programmer's hunch that the problem is pertinent to the concern is correct, and when the aspect cleanly captures the concern. When these conditions do not hold, this strategy may lead to a drop in programmer performance.

Although gathered at an early stage in the evolution of aspect-oriented programming, these empirical results can help evolve the approach in several ways. First, the results can help builders of cross-cutting modularity techniques, such as aspect-oriented programming and the closely related subject-oriented programming [9], improve the usefulness and usability of the techniques. The results can also help bridge to another useful form of empirical study—longer running industrial-based case studies—by helping potential early adopters of the technology determine whether the technique is suitable to address some of their development problems. Finally, software engineering researchers may build on our methods and results to continue experimental studies of both aspect-oriented programming and other separation of concern techniques.

We begin by describing the essential features of AspectJ (Section 2). We then describe our experimental method and present the results of running each experiment (Section 3). In Section 4, we describe the insights into aspect-oriented programming that arise from the studies. Section 5 critiques the study format used and Section 6 discusses related work on studies investigating the effect of program structure on programming tasks. We conclude with a summary of the paper (Section 7).

2 ASPECTJ

AspectJ consists of a slightly modified form of Java, called `JCore`, for expressing the core functionality of a program, plus a set of aspect languages. In version 0.1 of AspectJ, two aspect languages are supported: `Cool` for expressing synchronization concerns, and `Ridl` for expressing remote data transfer and method invocation concerns.

`JCore` is essentially identical to Java, save that the keyword `synchronized` and the `wait`, `notify` and `notifyAll` methods have been removed to ensure appropriate separation of synchronization concerns.

`Cool` encapsulates the synchronization aspect of AspectJ programs. A snippet of code called a coordinator is written when coordination is desired for a particular class or set of classes. This coordination can be described on a per-instance or a per-class basis. Each coordinator describes the synchronization on and between methods of the class or classes it coordinates via three constructs: `selfex`, `mutex`, and `guards`. A `selfex` specification on a method means that only one thread can concurrently execute that method. A `mutex` specification on two or more methods means that if one thread were executing one of these methods, no other thread could concurrently execute any of the other methods. A `mutex` specification does not imply a `selfex` specification. Finally, the `guard` construct is provided for more complicated, potentially dynamically-changing synchronization relationships; they permit the specification and enforcement of essentially arbitrary pre- and post-conditions on the execution of a method. A given method is permitted to simultaneously have a `selfex` specification and multiple `mutex` and `guard` specifications. Version 0.1 limits each class to a single coordinator.

`Ridl` allows the specification of remote interfaces for classes and the data transfer behaviour to be used when these remote interfaces are invoked. Only methods specified in the remote interface can be invoked remotely. The remote interface specifies how the input parameters and return value for each method should be transferred over a network: by passing a copy of the remote object, or by passing a global reference to the remote object. Furthermore, the fields of any of these objects can be individually specified as being passed by copy, passed by reference, or skipped altogether. The rationale behind such specifications is that the implementor of

[2] Version 0.2 of AspectJ provides more general purpose support for programming cross-cutting concerns [1].

the remote interface has knowledge of the way these objects are being used, and would therefore know whether it were costlier to make multiple communications to a remote host, or to copy an object all at once. `Ridl` version 0.1 is built on top of Java's Remote Method Invocation (RMI) protocol.

Each `JCore` class, `Cool` coordinator, and `Ridl` remote interface specification must reside in a separate file. At compile time, a tool called an aspect weaver combines these separate specifications into a set of Java classes, which are then compiled to produce executable bytecodes. A tool similar to `make` performs weaving and compilation only on those classes that are impacted by changes to the source files.

Figure 2 shows a portion of a class and its attendant coordinator and remote interface that were used in our studies. Since `addBook()` alters `bookCount` while `numBooks()` returns its value, the two methods should not be called concurrently. The coordinator (`Query.cool`) contains a `mutex` specification to ensure this condition. The remote interface (`Query.ridl`) indicates that the book to be added to the query should be passed by copy, while the library from which it came should be passed by reference.

Query.jcore

```
public class Query {

  Hashtable books;
  int bookCount = 0;

  public void addBook( Book b,
                       Library source ) {
    if( !books.containsKey( b ) ) {
      books.put( b, source );
      bookCount++;
    }
  }

  public long numBooks() {
    return bookCount;
  }
}
```

Query.cool

```
coordinator Query {
  mutex{ addBook, numBooks };
}
```

Query.ridl

```
remote Query {
  void addBook( Book b,
                Library source ) {
    b: copy;
    source: gref;
  }
}
```

Figure 1: Snippets of AspectJ Code

3 EXPERIMENTS

Our main goal in conducting these experiments was to better understand how the separation of concerns provided by aspect-oriented programming affects a programmer's ability to accomplish different kinds of tasks.

General Format

Each experiment consisted of six sessions: in three sessions, participants worked with AspectJ; in the other three, participants worked with a control language. Each session began with training time to allow the participants to familiarize themselves with the environment and the language(s) they were to use. We also gave the participants some refresher material on synchronization and distribution. The participants were then given ninety minutes to tackle the assigned tasks. Two computers were available for use in each session. The participants were graduate students and professors of computer science, and an undergraduate in computer engineering.

We videotaped the ninety minute sessions during which participants worked on tasks; the participants were asked to think-aloud during this time. An experimenter was present during the session and was available to answer questions about the programming environment. At thirty minute intervals, or after each task was completed, the experimenter stopped the participants and asked a series of questions:

- What have you done up to now?
- What are you working on?
- What significant problems have you encountered?
- What is your plan of attack from here on?

The same basic system—a digital library—was used in each experiment. The library had two main actors: readers and libraries. 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.

The library system was initially written in two languages, AspectJ (with `JCore` and `Cool`) and Java, the control language. These initial implementations were used in the program debugging experiment. For the change experiment, a distributed version of the system was then implemented in AspectJ (using `JCore`, `Cool`, and `Ridl`), and in Emerald. To more fairly compare Java and Emerald with AspectJ, synchronization lock classes in each language similar to the synchronization mechanisms of AspectJ were provided to the participants.

Our experimental design was a refinement of a design used to conduct a small pilot study that compared the ease of creating AspectJ programs with Java programs.

Experiment 1: Ease of Debugging

The intent of this experiment was to investigate whether programmers working with aspect-oriented programming were able to more quickly and easily find and fix faults in a multi-threaded program. Our hypothesis was that programmers working with the aspect-oriented programming language, AspectJ (`JCore` and `Cool` components), would be able to more quickly and easily identify the cause of and corrections for errors than programmers working in Java, the control language.

Three synchronization errors were introduced into the digital library code. Pairs of programmers, knowledgeable in multi-threaded programming techniques and object-oriented programming, then attempted to correct the faults.

Format
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 faults, and on-line documentation. The teams were asked to fix each fault sequentially. All participants were told that the errors were due to incorrect synchronization within the program.

The faults 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 fault, only one reader would make requests while the others remained idle. The participants had to remove per-class self-exclusive coordination on the `run()` method of the `Reader` class so that more than one reader (each in a separate thread) could run. In the second fault, multiple readers would make requests but the system would eventually 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. Removing per-object self-exclusive coordination on the `remoteSearch()` method of the `Library` class removed the deadlock condition. The third fault allowed more than one reader to check out the same book from the same library. To correct this, the participants had to add 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.

Results
In both the AspectJ and Java groups, all pairs of participants were able to find and correct all three of the faults. We analyzed videotapes of the sessions to extract both qualitative and quantitative data elements[3] such as the time taken, the number of builds, and the participant's views. We first discuss each data element in isolation, and then correlate and summarize the results.

Time The times required to correct each of the three faults are shown in Figure 2a. In this (and following) figures, a bar, shaded according to the language being used, is shown for each participant and for each assigned task. From Figure 2a, we can see that the largest difference in completion times was with respect to the first fault: the AspectJ teams clearly repaired the fault faster than the Java ones. For the second and third faults, there was a smaller difference.

Switching Between Files We examined the number of times the pairs switched the file they were examining to determine if the AspectJ users were affected by the coordination specification residing in a different file from the rest of the code. Figure 2b shows that the AspectJ pairs typically made fewer file switches than the Java group for fault 1, more for fault 2 and slightly less for fault 3.

[3] The raw data for these elements is available on a web page [19].

Instances of Semantic Analysis Figure 2c highlights the difference in the number of instances of semantic analysis over the 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...". The data indicates that the Java pairs more often analyzed the behaviour of the code than the AspectJ pairs. 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 fault:

> A: ...we *know* it's in the COOL file...
> B: But we have to know what they *do* before changing anything. —*AspectJ Pair 2*

Builds Overall, the AspectJ and Java pairs spent roughly equal time in building and executing their program. The additional time required for weaving AspectJ was negligible. The number of builds per fault ranged from one to nine.

Concurrency Granularity The Java users specified synchronization constraints by inserting statements about lock objects into methods. Working at the statement level meant that the Java users could attempt to synchronize parts of methods. To alter the concurrency granularity, the AspectJ pairs would have had to change the structure of the existing methods. To determine the instances when Java users considered finer granularity locking, we noted when the users attempted to move locks around within a method. Only one Java pair investigated locking granularity in the first fault, one in the second, and two in the third. None of the AspectJ participants questioned the granularity imposed by `Cool`.

Participants' Comments At the end of the sessions, two of the three AspectJ pairs expressed enthusiasm in support of separating out the coordination code, and felt that the separation directly contributed to their ability to solve the faults.

> 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. —*AspectJ Pair 2*

The other group, however, felt that `Cool` provided a handy way of summarizing coordination of and between methods, but were unhappy with the physical separation of the coordination code.

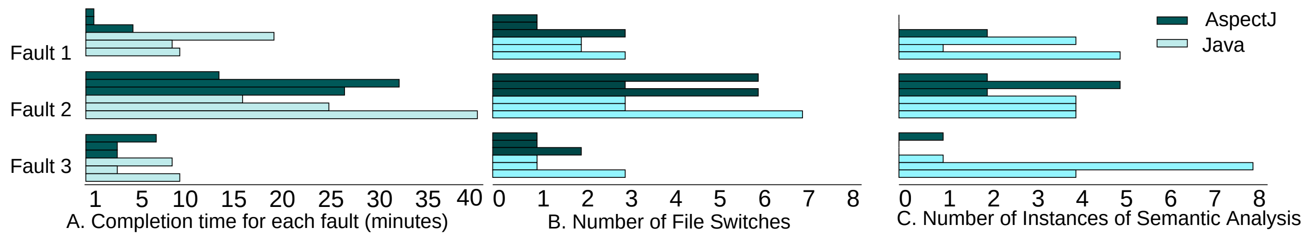

Figure 2: Debugging Results

> 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 faults to synchronization issues in which case you don't really have to understand the code.—*AspectJ Pair 3*

This pair would have opted instead for the `Cool` code to have been inserted in pertinent places throughout the code so that the programmer could see in one glance both the coordination and the method code at the same time. Interestingly, although this pair perceived that the separation provided by `Cool` caused them to look at many files to gain context, this pair switched less between files in total than any of the Java pairs.

Analysis of Results The three debugging tasks can be categorized two ways: according to addition or deletion of concurrency functionality, and according to localized or non-localized reasoning requirements. We say a fault required localized reasoning if the code responsible for the fault was modularized (e.g., part of one class). Non-localized reasoning meant participants would have to look across modularity boundaries for the problem.

The first fault's solution required localized reasoning and deletion of synchronization. In this fault, the AspectJ pairs were able to solve the fault faster than the Java pairs. They did so with fewer file switches, and fewer instances of semantic analysis. This points out that the AspectJ pairs were able to more quickly isolate and remedy the problem causing the fault.

Solving the second fault required non-localized reasoning and involved deleting synchronization. For this fault, the AspectJ pairs were somewhat faster than the Java pairs, and completed the task with slightly fewer file switching and marginally fewer instances of semantic analysis. Clearly, the AspectJ participants did not benefit as much from the use of `Cool` as they did in the first debugging task when the reasoning required for the problem was more localized.

Fixing the third fault required localized reasoning and involved adding synchronization. In this fault the AspectJ pairs generally finished faster and performed somewhat fewer file switches. Two of the Java pairs performed significant numbers of instances of semantic analysis, while two of the AspectJ pairs performed none. This is possibly because the Java participants had to perform analysis to understand how to add locking functionality, whereas using the `Cool` syntax required less analysis.

To summarize, when the solution to a problem required localized reasoning, `Cool` helped programmers focus their efforts. However, when the solution required non-localized reasoning, `Cool` did not provide as dramatic a benefit. This was regardless of whether functionality was being added or deleted.

Experiment 2: Ease of Change

The intent of this experiment was to investigate whether the separation of concerns provided in aspect-oriented programming enhanced a programmer's ability to change the functionality of a multi-threaded, distributed program. Our hypothesis was that the AspectJ combination of `JCore` for the component programming, `Cool` for synchronization, and `Ridl` for specifying data transfers would make it easier to change such programs compared to a similar program written in Emerald. Emerald was chosen as the control language because it is an example of an object-oriented language that integrates explicit, but not separate, support for distributed, synchronized programming.

Format

In this experiment, the participants worked alone. They were asked to address each of three change tasks sequentially. In the first task, participants were asked to add the ability for a reader to check books back into the library after checking them out. The solutions to this problem generally involved adding a method to check books back in, synchronizing that method, and calling it from somewhere within the main program loop. The second task was to assign one library to randomly reject a reader's request to check out a book. Adding this functionality required the determination of a library to make the denial decision, and the addition of a check in the main library code to ask that library if the reader's request for a check-out should be granted. The third task involved enhancing the performance of the code. Here the participants could find and fix any performance lag. However, to try to direct their approach, we seeded an inefficiency into the code: readers read the book byte-by-byte, requiring many messages to be sent if the book were located remotely. Enhancing the performance in this case meant ensuring that the appropriate book object was on the same host machine as the reader reading it. Because the third task was intended to be more open-ended and exploratory than the other two, we treat it separately in the discussion of results below.

The code in the solutions of each of the three tasks did not affect each other.

Results
We examined the performance of the participants by examining their approach to solving the problem. In particular, we analyzed the video-tape for such data elements as the time spent analyzing the code base versus the time spent writing their solutions. We considered both the absolute time and the proportion of the total time spent on each activity. We also looked at the pattern of activities over time. Finally, we examined the code written by the participants. We describe these data elements individually [19] before synthesizing the results.

Time Figure 3a shows the completion times for the six participants for the first and second tasks. The Emerald participants typically had faster completion times than the AspectJ participants.

Portion of Time To investigate what could account for the time differences between the Emerald group and the AspectJ group, we examined how the participants spent their time. Figures 3b and 3c show, as a percentage of total time, how much time was spent coding and analyzing for each activity for each participant. The typical percentage of time spent on coding was slightly greater for the AspectJ trials, while that spent on analysis was greater for the Emerald trials. The remaining percentage of time was spent on a combination of compiling and running the program. The AspectJ participants spent slightly more absolute time observing the program run.

Patterns of Activities Emerald participants typically began their tasks with extended periods of analysis while AspectJ participants typically began extensive coding attempts with little or no prior analysis.

Code Written The AspectJ participants wrote between 50 and 150 lines of code, of which two to six lines were `Ridl` code, and two to three lines were `Cool` code. The Emerald participants wrote between 50 and 80 lines of code, of which two to four lines were synchronization code, and one to three lines pertained to the movement of objects.

Task 3: Performance Enhancement While each Emerald participant successfully made at least one modification to the program that led to a performance enhancement, only two AspectJ participants had sufficient time to attempt this task, and only one of these successfully improved the performance.

Since no support is provided for object replication, the `Ridl` specifications can violate the pass-by-reference or pass-by-copy semantics required by the `JCore` code. The participant who unsuccessfully attempted this task encountered and recognized this difficulty.

Since `Ridl` version 0.1 is implemented on top of Java RMI, copying an object requires that its class implement the `Serializable` interface; to pass a global reference to an object requires that its class provide a remote interface. This causes a serious catch-22 for standard Java library classes that implement neither. The alternative to using such library classes is to encapsulate their instances within “remoteable” versions; however, the `JCore` code would then need to specify explicitly that the “remoteable” version be used. Although we provided such “remoteable” versions for a few classes that we suspected would be needed to complete tasks 1 and 2, the open-ended nature of task 3, combined with the expensive nature of re-implementing most of the standard Java library, prevented us from providing a sufficiently complete set of such versions. Both AspectJ participants encountered problems when the remote interface specifications they attempted involving library classes either failed to compile or caused unexpected run-time exceptions.

Participants’ Comments When the two AspectJ participants who attempted the performance-enhancing task were asked at session-end what difficulties they had encountered, they both believed that the amount of code analysis required to express a concern in `Ridl` was a factor. One of these participants noted that the separation between `Ridl` and `JCore` was not as clean as desired.

> I get the feeling that `Cool` is pretty close to capturing synchronization but that `Ridl` has a way to go…it’s too meshed with Java—*AspectJ Participant 3*

This participant believed that it may be more difficult to separate object mobility issues from the core functional code than separating synchronization issues. The participant characterized object mobility issues as the location of an object at a particular execution point and how an object is passed into a method. These participants claimed that in order to understand how objects were moving around in the system, it was necessary to thoroughly understand the core semantics that supported the `Ridl` file.

Analysis of Results The Emerald participants were able to implement more, faster than the AspectJ participants.

The patterns of activity for the AspectJ participants showed a heavy emphasis on coding quite early in their tasks, as compared to those for the Emerald participants. This may point to the fact that massaging the `Ridl` code seemed like a quick way to solve object-mobility problems when, in fact, it was not. Interestingly, the AspectJ participant who successfully attempted task 3 did spend more of their time on analysis of

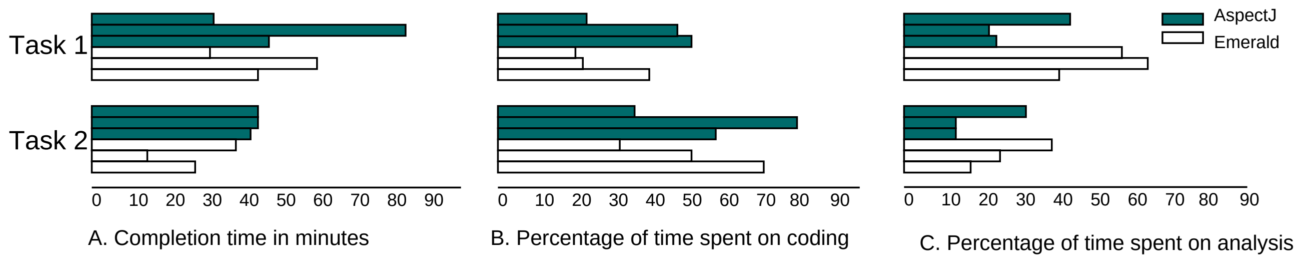

Figure 3: Change Results

the core semantics than on coding, while the AspectJ participant who unsuccessfully attempted task 3 showed the opposite distribution. This observation lends some credence to their claims that in-depth analysis of the core semantics is required to correctly express a concern in `Ridl`.

The fact that AspectJ participants spent slightly more absolute time observing the program run could also suggest they perceived that tinkering with the `Ridl` code and watching the program run would keep them from having to deal with the core semantics of the program.

4 INSIGHTS FROM COMBINED ANALYSIS

Combined analysis of the results of both the debugging and change experiments provides two key insights into characteristics of aspect-oriented programming that may significantly affect the usefulness and usability of the approach.

Aspect-Core Interface Matters

Separating concepts into different pieces of code does not imply independence of those code snippets. Interaction between the separate pieces is needed to create the behaviour of the system. In this discussion, we refer to the boundary between code expressed as an aspect and the functionally-decomposed core code as an aspect-core interface.

Our participants noted the effect of the aspect-core interfaces on the tasks that they performed. For the most part, the AspectJ participants found that the effect of the `Cool` code on the `JCore` code had a well-defined scope: we refer to this as a *narrow* aspect-code interface. The narrow aspect-code interface allowed participants to understand the `Cool` code without inferring or analyzing extensive parts of the core code. Figure 4 illustrates the typical analysis necessary; only those methods explicitly mentioned within the coordinator are affected by its synchronization specifications.

The AspectJ participants in the second experiment had more difficulties with the aspect-core interface between `Ridl` and `JCore`. This interface is wider, meaning it is necessary to look at both the aspect code and large chunks of the core code to understand the aspect code. Figure 5 illustrates this point. Because `Ridl` alters the nature of the data being transferred, the potential impact of the remote interface specification extends beyond the method explicitly mentioned therein to include the transitive closure of methods using that method. An extensive analysis is required to ensure that constraints in place within the core code are not violated by the remote interface. As one participant noted,

> `JCore` and `Ridl` interact more than I would like …you can't ignore `JCore` code in your `Ridl` semantics.… I was very often looking at the `JCore` implementation so that I could decide what was wise to do in `Ridl`.—*AspectJ Participant 3*

The partial separation provided by `Ridl` may have actually hindered the performance of the change tasks by the AspectJ participants. Partial separation should thus not be considered to necessarily bring partial benefit.

By paying careful attention to the design of aspect-core interfaces, builders of aspect-oriented programming environments may be able to help a programmer focus more easily on code relating to a task, aiding the programmer's ability to complete some tasks. Tool support, such as an impact analyzer, might also help a programmer cope with different kinds of aspect-core interfaces. While we have provided a high-level definition of aspect-core interface, this remains vague as significant work is required to understand the true nature of the coupling between aspects and core code.

Aspects May Alter Task Strategies

All AspectJ participants tended to first consider the aspect files for solutions to perceived coordination or data transfer problems. In the debugging experiments, this strategy was successful, since all of the solutions were solely programmed in the `Cool` files. However, in the change experiment, it was necessary to understand, and sometimes change, files containing core functionality to complete an assigned task.

As we discussed earlier, the AspectJ participants in the change experiment typically commenced coding much ear-

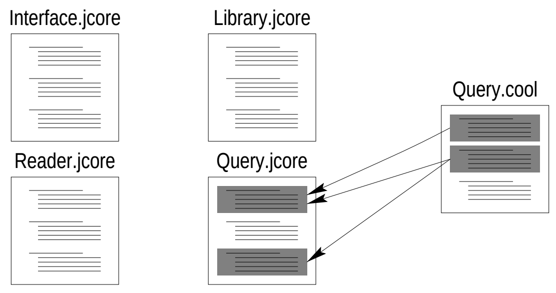

Figure 4: Localized Semantic Analysis with `Cool`

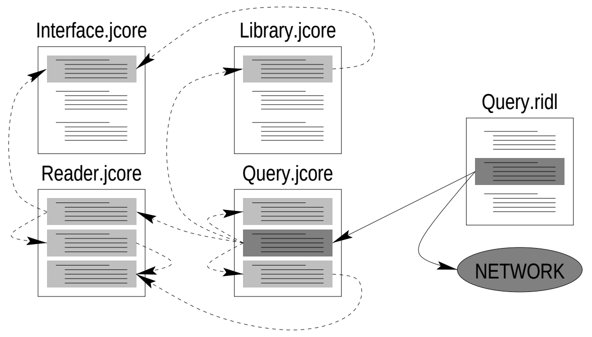

Figure 5: Non-localized Semantic Analysis with `Ridl`

lier in a task, and spent more time coding overall, compared to the participants in the control group. We observed that the participants initially assumed they could solve the data-transfer problems by looking at and massaging the `Ridl` files. This presumption may have distracted the participants from analyzing the core functionality code to the extent that was necessary for the task.

The presence of aspect code, then, may alter the strategies programmers use to approach a task. Specifically, programmers may try to tackle problems first in the aspect code, which may be shorter and simpler, rather than gaining a suitable knowledge of both the core functional and aspect code. This behaviour alteration could affect the initial usability, and acceptance, of an aspect-oriented programming approach. It also indicates that programmers may have difficulty in performing tasks perceived to be associated with aspect code when the aspect code does not suitably encapsulate a concern. Longer-running studies where programmers are able to gain more experience with aspect-oriented approaches are needed to further investigate this second point.

5 EXPERIMENTAL CRITIQUE

Our goal in undertaking an early assessment of aspect-oriented programming was to gather information that could be helpful in directing the evolution of the technology. In support of this goal, we wanted to gather data about both the performance, and the experience, of programmers using the approach. We chose an exploratory study format in which we compared the results of a small number of programmers using an aspect-oriented approach with others using a control language for several reasons:

- the control groups provided a basis on which to assess the performance of the AspectJ groups,
- the cost of running and analyzing a trial was high so we wanted to balance cost with the maturity of the technology,
- the pool of potential participants and the amount of time available from each participant was limited, and
- we chose to forfeit some precision in measurement in favour of realism [16].

Within these constraints, we took several steps to ensure our results had some validity. To achieve internal validity, we provided the different groups—aspect-oriented versus non-aspect-oriented—with as similar support as possible, limiting variances, as much as possible, to the features of interest. For instance, the lock constructs we provided for use in Java and Emerald provided a mechanism similar to that available in AspectJ.

To address construct validity, we gathered data from multiple sources. One source was the qualitative statements made by the participants during the taped interviews; the other sources were the data analyzed from the tapes. Sometimes the data from the multiple sources was corroborative, other times it was contradictory. Corroborative data strengthened the result under discussion: contradictory data weakened the result.

The reliability of our experiments was high with respect to the procedures we followed in conducting the experiments and analyzing the data. However, as expected, the skills of the participants varied greatly as finding participants with Java (or Emerald) experience, concurrent programming, and, in some cases, distributed programming was difficult.

The variability in the skills of participants and the modest number of participants limits the generalizability of our results. We chose to make this trade-off because, in these exploratory studies, we were interested in *how* the participants worked with the approach; the quantitative data supported the analysis of the qualitative data.

The external validity of the experiments is also affected by the problems we chose as a basis for the experiment and the limited training provided to the participants. The faults seeded into the system for the debugging experiment, for instance, were all synchronization problems that could be solved by altering `Cool` code. We informed the participants that synchronization faults had been seeded into the program; AspectJ participants may thus have been pointed towards the `Cool` code. The performance of all of the participants may also have been affected by being asked to work with either new languages (the AspectJ participants), or with particular constructs (Java and Emerald) introduced to provide a basis of similarity between the languages. Scholtz and Wiedenbeck have shown that programmers experience a drop in performance and their solution process is disrupted when using an unfamiliar programming language [21]. All of our participants likely experienced this effect in differing degrees; our experimental method did not allow us to explicitly quantify or qualify the differences.

The limitations in our studies could be overcome by refining and expanding the experimental method. However, the cost entailed in conducting more controlled experiments must be weighed against the development curve of the technology being studied. Our exploratory study format has provided insights useful for researchers building aspect-oriented programming environments at this early stage of the technology.

6 RELATED WORK

Our work in evaluating how the structuring of aspects impacts programming tasks continues a line of inquiry that began with the introduction of structured programming and data abstraction techniques. Curtis et al. [6] synthesized two themes from this body of work that considered the effect of control structures and data structures on programming tasks: "structuring the control flow assists programmers in understanding a program" [p. 1095], and "data structuring capability and data structure documentation may be strong determinants of programming performance" [p. 1096].

Aspect-oriented programming builds on this earlier structuring work providing increased support for modularizing the code. Fewer studies have been conducted on the impact of modularity choices on programming tasks.

Rombach reports on controlled experiments that compared the ease of maintaining systems developed in the LADY distributed system language with similar systems implemented in an extended version of Pascal [20]. These experiments showed that the additional structuring information available in the LADY implementations aided the isolation of program faults, and that less rework was required in requirements, design, and coding when adding a new feature. Although the kind of additional structuring provided in AspectJ is different than in LADY—LADY provides support for hierarchical structuring of distributed system solutions—the faster times for solving seeded problems in the AspectJ implementation compared to the Java implementation may also be a result of additional structuring information helping fault isolation. We did not see a similar increase in performance for adding new features into an AspectJ program.

Korson and Vaishnavi conducted an experiment to investigate the effect of modularity on program modifiability [13]. They found evidence to suggest that a modular Pascal program was faster to modify than a non-modular version when one or more of three conditions held: modularity was used to localize change required by a modification, existing modules provided some generic operation that could be used in implementing a modification, or a broad understanding of the existing code was required to perform a modification. In our debugging experiment, the localization of concurrency information in an aspect module may have eased the task of adding synchronization information into the system compared to the Java programmers. When the aspect code does not cleanly modularize a concern, as was the case in the change experiment, we did not see a benefit.

Boehm-Davis et al. investigated four questions relating to the effect of program structure on maintenance activities ranging from the role of structure when modifying a system to whether structure affected the subjective reactions of programmers to a system [4]. The study involved both student and professional programmers making either simple or complex modifications to programs written in either an unstructured style, a functional style, or an object-oriented style. The study showed that the ability of a programmer to abstract information from code was important in the modification process, and that this ability was affected by the structure of the system and the programmer's background. We have discussed how the nature of the separation provided by aspect code can affect a programmer's ability to understand source code during the performance of a task. Our experimental format did not support an investigation of how system structure and programmer's background might impact aspect-oriented programming.

In providing a means of localizing cross-cutting information, aspect-oriented programming shares some features of program slicing [22]. An empirical study by Law of the effect of slicing on debugging tasks found that "the experimental group who [had] obtained the knowledge of program slicing only took fifty-eight percent of the control group's time to debug a one-page C program [14]" [15, p. 43]. Slices differ from aspects in several ways: for instance, slices are typically computed on-demand rather than appearing explicitly as part of the program text, and slices generally describe the inter-dependencies of program statements rather than localizing programming concerns. Further studies are needed to understand how each of these different cross-cutting mechanisms impacts the program development cycle.

7 SUMMARY

We report on two exploratory experiments we conducted to the increased program modularization provided by AspectJ, an example of the emerging aspect-oriented programming approach. In these experiments, we compared the performance and experience of participants working on two common programming tasks: debugging and change. Some participants worked with AspectJ; other participants used an object-oriented control language.

We noted that in the first experiment the AspectJ participants were able to finish the tasks faster than the participants using Java, the control language. The Java participants performed more semantic analysis, and switched the file they were viewing more often, than the AspectJ participants. In the second experiment, the AspectJ participants required more time to complete tasks than the participants using Emerald, the control language. Analysis of the AspectJ participants' activities showed that these participants typically spent more of their time coding their solutions and less of their time analyzing the existing code base than the Emerald participants.

These results suggest two key insights into aspect-oriented programming. First, programmers may be better able to understand an aspect-oriented program when the effect of aspect code has a well-defined scope. Second, the presence of aspect code may alter the strategies programmers use to address tasks perceived to be associated with aspect code.

Builders of aspect-oriented approaches may benefit from considering these two key insights. The insights suggest particular characteristics an aspect language may need to exhibit in order to ease the performance of programming tasks. Specifically, aspect languages should enable the writing of aspect code that has a well-defined scope of effect on core functional code, and that suitably encapsulates a concern. These insights may also apply to other cross-cutting modularization techniques, such as subject-oriented programming.

ACKNOWLEDGEMENTS
We thank the Xerox PARC Embedded Computation Area group for their comments on and involvement in the studies, the use of the AspectJ weaver, and the fast responses to solving the few problems with the environment that occurred. We also thank Robert Rekrutiak and Paul Nalos for their contributions to experiment setup and design, Gregor Kiczales and Robert Bowdidge for their comments on an earlier draft, and our anonymous experimental participants.

REFERENCES

[1] AspectJ web page, 1998. http://www.parc.xerox.com /spl/projects/aop/aspectj/.

[2] V. Basili. The role of experimentation: Past, current, and future. In *Proc. of the 18th Int'l Conf. on Software Engineering*, pages 442–450, 1996.

[3] A. Black, N. Hutchinson, E. Jul, and H. Levy. Object structure in the Emerald system. *ACM SIGPLAN Notices*, 21(11):78–86, Nov. 1986.

[4] D. Boehm-Davis, R. Holt, and A. Schultz. The role of program structure in software maintenance. *International Journal of Man-Machine Studies*, 36(1):21–63, Jan. 1992.

[5] R. Corp. UML summary. Web document: http: //www.rational.com/uml/html/summary/.

[6] B. Curtis, E. Soloway, R. Brooks, J. Black, K. Ehrlich, and H. Ramsey. Software psychology: The need for an interdisciplinary program. *Proceedings of the IEEE*, 74(8):1092–1106, 1986.

[7] E. Dijkstra. *A Discipline of Programming*. 1976.

[8] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. 1996.

[9] W. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects). In *Proc. of OOPSLA '93*, pages 411–428, 1993.

[10] J. Irwin, J. Loingtier, J. Gilbert, G. Kiczales, J. Lamping, A. Mendhekar, and T. Shpeisman. Aspect-oriented programming OS sparse matrix code. In *Proc. of Int'l Conf. on Scientific Computing in Object-Oriented Parallel Environments*, pages 249–256, Dec. 1997.

[11] K. Jensen and N. Wirth. *Pascal: User Manual and Report*. LNCS 18. 1974.

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97—Object-Oriented Programming, 11th European Conference*, LNCS 1241, pages 220–242, 1997.

[13] T. Korson and V. Vaishnavi. An empirical study of the effects of modularity on program modifiability. In E. Soloway and S. Syengar, editors, *Proc. of the First Workshop on Empirical Studies of Programmers*, pages 168–186, 1986.

[14] R. Law. Evaluating the program slicing technique. *SIAST Today*, 4(6):6, June 1993.

[15] R. Law. An overview of debugging tools. *ACM Software Engineering Notes*, 22(2):43–47, Mar. 1997.

[16] J. McGrath. Methodology matters: Doing research in the behavioral and social sciences. In *Readings in Human-Computer Interaction: Toward the Year 2000*, pages 152–169. 1995.

[17] A. Mendhekar, G. Kiczales, and J. Lamping. RG: A case-study for aspect-oriented programming. Technical Report SPL97-009 P9710044, Xerox PARC, Feb. 1997.

[18] G. Murphy, R. Walker, and E. Baniassad. Evaluating emerging software development technologies: Lessons learned from assessing aspect-oriented programming. Technical Report TR-98-10, UBC, Dept. of Computer Science, 1998. Submitted to IEEE TSE.

[19] Raw experimental data web page, 1998. http://www.cs.ubc.ca/labs/se/projects/aop/.

[20] H. D. Rombach. Impact of software structure on maintenance. In *Proc. of the Int'l Conf. on Software Maintenance*, pages 152–160, 1985.

[21] J. Scholtz and S. Wiedenbeck. The use of unfamiliar programming languages by experienced programmers. In *People and Computers VII: Proceedings of HCI'92*, pages 45–56, 1992.

[22] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4), 1984.