Managing Crosscutting Concerns During Software Evolution Tasks: An Inquisitive Study

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

Christa Schwanninger and Michael Kircher
Siemens AG, ZT SE 2
Otto-Hahn Ring 6, 81739,
Munich, Germany
{christa.schwanninger, michael.kircher
@mchp.siemens.de}

ABSTRACT
Code is modularized for many reasons, including making it easier to understand, change, and verify. Aspect-oriented programming approaches extend the kind of code that can be modularized, enabling the modularization of crosscutting code. We conducted an inquisitive study to better understand the kinds of crosscutting code that software developers encounter and to better understand how the developers manage this code. We tracked eight participants: four from industry and four from academia. Each participant was currently evolving a non-trivial software system. We interviewed these participants three times about crosscutting concerns they had encountered and the strategies they used to deal with the concerns. We found that crosscutting concerns tended to emerge as obstacles that the developer had to consider to make the desired change. The strategy used by the developer to manage the concern depended on the form of the obstacle code. The results of this study provide empirical evidence to support the problems identified by the aspect-oriented programming community, and provide a basis on which to further assess aspect-oriented programming.

Keywords
Empirical study, aspect-oriented programming, software evolution

1. INTRODUCTION
Code is modularized to make it easier to read, change [1], and verify, amongst other reasons. Aspect-oriented programming approaches [2, 3, 4, 5, 6, 7, 8] extend the kind of code that can be modularized by providing support for modularizing crosscutting code.

The aspect-oriented approaches were developed based on certain instances of crosscutting code, such as code associated with distribution [9], synchronization policies [10] and some kinds of features [7]. To our knowledge, no independent empirical studies have been undertaken to consider the range of crosscutting concerns that software developers would find beneficial to modularize, nor how those software developers are currently managing those concerns in existing systems. This paper helps fill this gap: It reports on a study in which eight software developers, each of whom was currently evolving a (different) system, were interviewed over a period of three weeks about their progression on a change task. Half of these participants were from industry and half were from academia. All were working on non-trivial changes to non-trivial systems.

Our analysis of the data collected during the study indicated that each of the developers had to consider at least one crosscutting concern. Crosscutting concerns arose when participants encountered obstacles in making their desired change. For instance, one participant encountered a memory allocation concern when trying to tailor an algorithm to a specific new purpose. Three strategies emerged for managing the obstacles: in some cases, the entire concern was changed, in other cases, the developers chose to work within the conventions of the concern, and in yet others, the choice was to alter the change task rather than to try to cope with the concern. The strategy chosen depended on how the concern interacted with the core code associated with the change.

This study and its results make two contributions. First, the results provide empirical evidence about the kinds of crosscutting concerns that impact software developers and the strategies developers use to cope with these concerns in existing systems. Second, the results provide a basis on which to compare whether the use of aspect-oriented approaches enables developers to better represent and work with crosscutting code. For example, if the use of an aspect-oriented approach eliminated the need to alter a change task in situations similar to those described in this paper, then that would be evidence of a benefit of the aspect-oriented approach.

We begin in Section 2 with a description of the study format. In Section 3, we report on the results of the study. In Section 4, we discuss the implications of the results. In Section 5, we review previous work related to this study. In Section 6, we summarize and conclude.

2. STUDY FORMAT
Our study was inquisitive [11]: we interviewed rather than observed or shadowed the participants. Over a three-week time period, we tracked the progress of participants on a change task to a system that they were evolving. In this section, we describe the details of the study. We discuss the limitations of the format in Section 4.

2.1 Background of Participants
The study involved eight participants with a broad range of backgrounds: four had years of programming experience in an

AOSD 2002, Enschede, The Netherlands

industrial setting, and were practicing software engineers from Siemens AG; four were graduate students from the University of British Columbia with a range of programming experience.

Only two of the participants were familiar with the concept of aspect-oriented programming prior to the study. One of these two participants was actively applying aspect-oriented programming ideas in the change task with which they were involved during the study, and the other had experience working with an aspect-oriented language. The rest of the participants had no knowledge of aspect-oriented programming at the start of the study.

To participate in the study, we required that a participant be working on, or have recently worked on, a program change task to a system for which they were neither an initial nor a principal developer. The rationale for this constraint was to ensure that participants would have to investigate the scope of the change since they had limited prior knowledge of the code base. Each participant in the study was working on a separate system.

Before commencing the study, participants were asked to provide the interviewer with a copy of the code they were working on to serve as a reference.

2.2 General Format

We organized the study as a series of interviews: each participant was interviewed three separate times, with each interview lasting up to an hour. The same interviewer conducted all interviews.

General guidelines to focus the interviews were prepared in advance. The specific questions that were asked depended upon the flow of conversation. Our goal was to determine four different pieces of information during each interview:

1. The program change task of the participant,
2. The approach of the participant to the task,
3. The approach used by the participant to determine which pieces of code needed to change, and
4. Whether the participant thought that the change was difficult to make and if so, why it was difficult.

To help focus the discussion, participants were asked to identify the portions of code that had, and that were, being changed. To keep track of these locations, we annotated the interviewer's copy of the source files.

All the interviews were audio taped and later transcribed.

2.3 Questioning Convention

A primary purpose of the study was to determine the kinds of crosscutting concerns that developers must consider in existing code bases. Rather than ask the participants directly about these concerns, we asked them questions about the change task on which they were working, and we attempted to glean concerns from their responses.

We took this approach for three reasons. First, most of the participants had never thought about crosscutting concerns. When we attempted to pose questions that directly asked about concerns, the participants were unable to understand the meaning of our questions. Second, there was a danger that the participants who did have some knowledge of this area would simply respond with popular crosscutting concerns like tracing, debugging, or distribution. Such a quick response might have hidden more task-related concerns. Third, when programmers are heavily involved in the details associated with a task, it takes time to ease them into coarser-grained thinking about their problem. Asking participants questions that they could answer readily from their own experience facilitated the gathering of data.

At the beginning of an interview, participants tended to talk about their change task in a detailed way. For example, one participant provided in-depth information about specific data structures used in the application. Typically, by the end of an interview, participants began to talk about their task at a more conceptual level. This shift in the level of detail enabled participants to consider higher-level questions, such as labels that they might use to describe the code they were examining, or methods that they had used to find the relevant portions of source for their task. The more conceptual level of thinking about the task enabled the interviewer to ask participants to think, between interviews, about the following question: If you could have any view of the code, what view would have helped you perform this task? This question was intended to help identify the portions of code that the participant would like to see modularized. To help make the question concrete, the interviewer provided sample answers, such as "all the code pertaining to the database system", or "all the code related to printing".

2.4 Method of Qualitative Analysis

To analyze the data, we examined the transcripts of the interview sessions and the annotated source code.

Our examination involved three passes of the transcripts. First, we perused the transcribed interviews to try to understand the range of responses. Second, we categorized the responses in terms of how the participants described the change they were attacking, and what they encountered while working on the change. Finally, we examined the responses for commonalties.

We also examined the annotated source code, looking at the form of the statements highlighted. We looked for commonalities in terms of syntax, semantics, or function. We also examined the code to try to determine whether the changes themselves could be characterized as belonging to a particular concern.

3. QUALITATIVE ANALYSIS

Most participants described their change task from two perspectives: a structural perspective, and an emergent obstacle-based perspective. Almost every participant at some point in an interview used the phrase: "Everything was going fine until …". We describe each of these two perspectives and then describe the results of an analysis on the change and obstacle code.

3.1 Straightforward Structural Perspective

Each participant began by providing detailed descriptions of the application problem domain and of the change. They described the field in which they were working, how their application fit into that field, and how their change fit into the application.

The participants' initial descriptions of their change task were in terms of easily identifiable structure in the code. Specifically, most participants described their change in terms of a particular data structure or a particular module in the code, such as "I was changing the components of a data structure", or "I was changing the methods related to the user interface". Describing the change in this way was straightforward, even though the code was often spread throughout the code base. The programmers could understand the purpose of the code and its context within the

structure of the application. They could point out portions of the code that corresponded to their change.

Only one participant—participant one—described crosscutting code as the target of the change. This participant was working in the area of aspect-oriented programming.

Table 1: Participants' task descriptions

Participant	Straightforward Structural view	Non-straightforward Obstacle View	Strategy
1	Moving particular computation to an aspect-like module	Synchronization, Performance	Within
2	Tailoring a matching algorithm for a specific purpose	Memory allocation	Change
3	Changing matrix calculation	Memory allocation	Around
4	Changing table representation	Implicit assumptions about data structure representations	Around
5	Changing packaging of user interface mechanism	Distribution, Tracing	Within
6	Changing the mathematical model applied	Security issues, Communication protocols, Hardware platform dependencies	Within
7	Changing printing look and feel	User Interface consistency, Printing speed	Change
8	Adding cancellation notification to an existing system	Multithreading, Behavioural consistency	Within

3.2 Non-straightforward Obstacle Perspective

After participants had described their change task, and after they had pointed out the locations in the code that they had to change, we asked them to consider if these were the only portions of code that had to change to complete the task. Invariably, they said "no". It was at this point that the participants revealed a set of obstacles that they had encountered when making the change.

Obstacles comprised portions of code that were relevant to the task but that also affected an underlying concern; this code was at the intersection of the core change and the broader concern. For example, participant eight was adding a feature to the system and had to ensure that the change was consistent with behavioural conventions. To make the change, the participant had to overcome the obstacles and to try to understand the entire underlying concern, the behavioural conventions, that led to the presence of that portion of code. Since that underlying concern was neither well-modularized nor well-documented, it was difficult to conceptualize and to reason about.

The participants used three strategies to cope with the obstacles:

1. *Change*: Alter the concern code to enable the change task.
2. *Within*: Understand, but do not change, the underlying concern associated with the obstacle sufficiently to make the change work within the concern.
3. *Around*: Completely alter the change task to account for the concern without understanding the concern.

Table 1 summarizes the program change tasks for each participant, the obstacles each encountered and the strategy each employed.

Change Strategy. Participants two and seven used the first strategy: they changed the relevant portions of the crosscutting concern to suit the change. For participant seven, this approach was facilitated by the fact that the changes were at the user-interface level, and thus were more visible during testing. Participant two's changes are discussed in more detail in Section 3.3.

Within Strategy. Participants one, five, six and eight used the second strategy. They worked hard to understand the effect of their code on the crosscutting concern that presented an obstacle to their change, and they worked within the conventions of the concern. Participant eight had to perform considerable testing to ensure the obstacle had been dealt with appropriately.

Around Strategy. Participants three and four used the third strategy: they each worked around the obstacle. They significantly rethought their original approach to their change task because they could neither adequately understand the obstacles, nor address the concern. Participant four, for example, ran into memory allocation problems after making what should have been a simple change to a table representation. After failed attempts to understand how the change affected the memory allocation for the application, a work-around was devised to trick the memory allocation portions of the source into thinking that the change had not been made.

3.3 Code Perspective

By examining the code associated with the changes and with the participants' comments, we learned more about how participants addressed the obstacles they faced. Our examination focused on the obstacle points; the locations at which the change task intersected the crosscutting concern. We discovered that there were certain patterns of interaction between the concern and the change code, and we determined that there was a correspondence between the patterns and the strategy the participant chose to address the obstacle.

Change Strategy. Code associated with participants who chose the first strategy, the change strategy, had a structural intersection point. Participants could identify, from the code related to the change, certain structures—types, objects, and computations directly related to those structures—as obstacles to their change task. Figure 1-A depicts this situation. These obstacle points, shown as black boxes, provided enough information about the broader concern to lead the participant along the outward reasoning arrow to the points of change, located in the broader concern shown in light grey. This situation was particularly true

for participant two. For this participant, the obstacle points were easily identifiable by the type of the data structure affected. Participant two was able to extrapolate that all functionality of a certain kind involving a particular type would have to be altered. It was then straightforward, though tedious, to make the changes.

Within Strategy. Code associated with participants who chose the second strategy, the within strategy, followed a behavioural pattern. Participant eight worked within computational conventions, and participant one had to work within a particular synchronization policy. The intersection of the change code and the behavioural concern code could not be as easily assessed as for the structural case above. As is shown in Figure 1-B, the obstacle points were implied. Comments alerted each of the participants to the presence of the obstacle, and gave them clues as to the existence and nature of the broader concern. Based on the comments, these participants had to examine the broader concern to understand the conventions of the concern. The participants then had to *reason inward* about how to change the core code to work within the broader concern. Their analysis techniques were ad hoc, and it was difficult for them to describe their approach. Essentially, they reported that they had to gain a general understanding of the code base in order to work within the concerns. Once they gained this understanding, they were able to identify portions of code that would allow them to reason inward about their specific change task.

Figure 2 shows the inward reasoning, and resultant code used by participant one. This participant was moving pre-fetching functionality within operating system code into a separate aspect-like module. Specifically, the participant wanted to migrate the

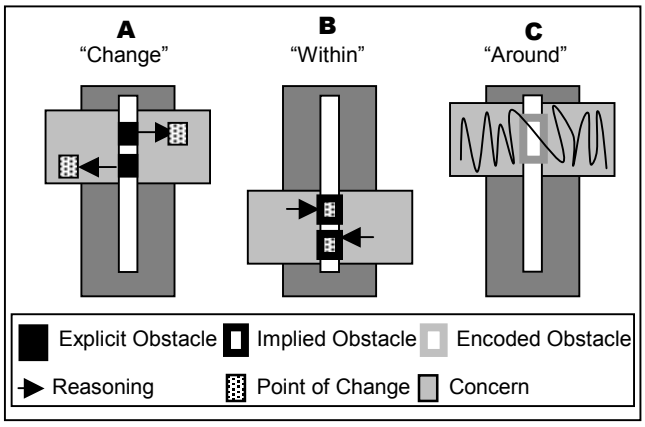

Figure 1. Obstacle types: Core-Concern Intersections

circled code in the black box on the left of Figure 2 to the pre-fetching module on the right. Based on previous knowledge, and on comments in the code, the participant knew that this change would impact synchronization in the system. Relevant synchronization code, shown in boxes A1 and B1, was identified by tracing up the call chain and pinpointing locking and unlocking code that could affect the code of interest. The developer had to reason inward from the synchronization concern to the core change. Synchronization code similar to that in boxes A1 and B1 had to be included in the new pre-fetching module (boxes A2 and B2) even though the code was not directly related to the core of the change. The inclusion of this code ensured that

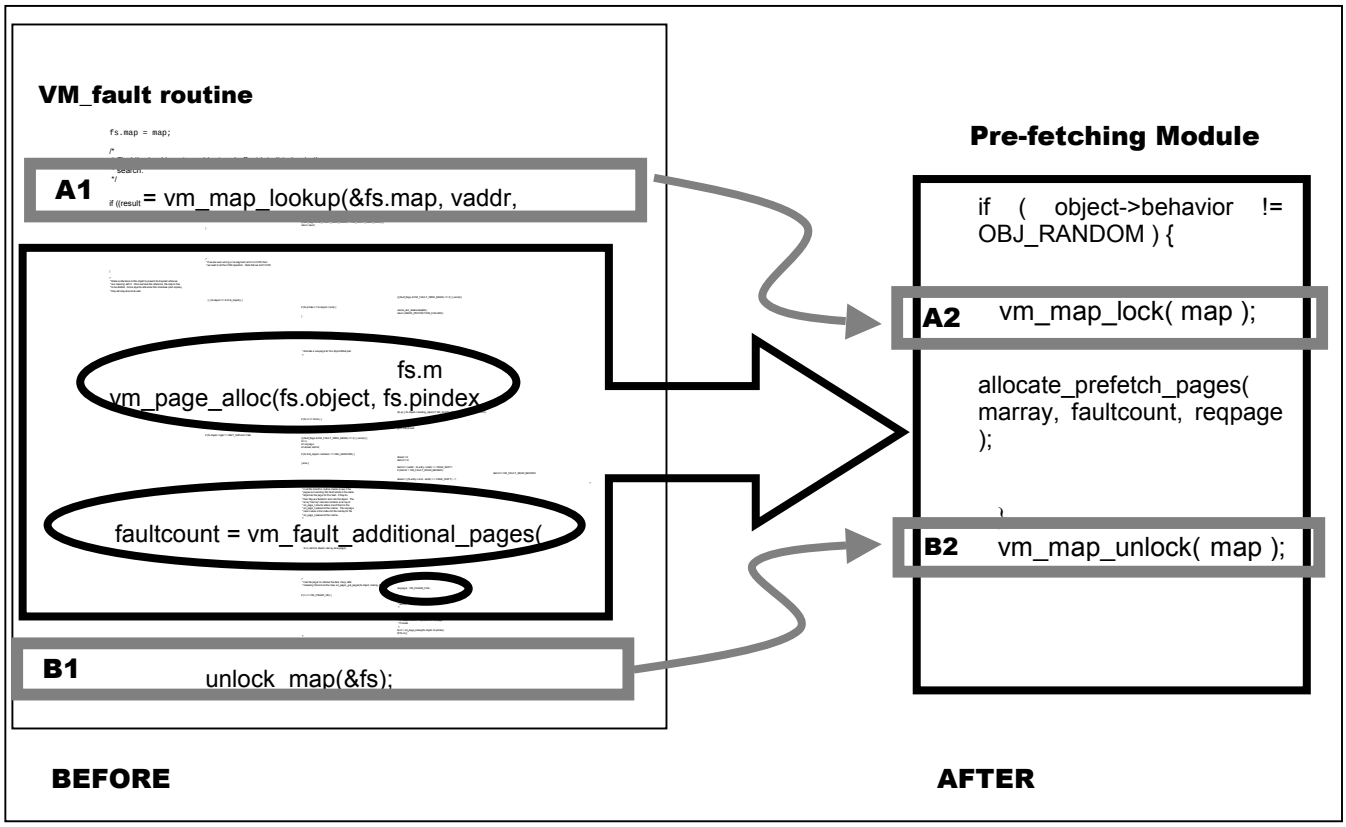

Figure 2: Code alterations show inward reasoning

the locking invariants encoded in the synchronization concern were maintained.

In all cases, participants were unable to cleanly determine when they had addressed all of the code related to their change. Our examination of their code yielded limited similarities about the nature of the concern code. In particular, for participant eight, the concern conventions could be gleaned by scanning for instances of a particular sequence of calls. When asked, participant one reported that this "sequence of calls" analysis might have been helpful. Participant one might also have been helped by information about a pattern of access to particular data structures.

Around Strategy. Obstacle code associated with participants who chose the third strategy, the *around* strategy, was dense. The code made ambiguous use of assumptions from around the code base and was thus subtle and difficult to reason about. Originally, these participants had wanted to change the relevant portions, rather than to avoid the code. However, when the change approach became too onerous, the participants were forced to work around both the obstacle and the concern code. It was typically unclear why particular data structures were altered in particular ways, and why the ordering of certain computations was important. For instance, the obstacle code encountered by participant four assumed that a data structure of a certain number of bytes (16) would be used. This number was relied upon heavily in the computations for allocating memory, but was never indicated explicitly.

The code in Figure 3 illustrates this situation in which it is neither mentioned in the comments, nor obvious from the code that the computations will be correct only when `DdNode` is equal to 16. When the participant wanted to change that value, it caused unpredictable results.

```
if (mem != NULL) {
/* successful allocation; slice memory */
   ptruint offset;
   unique->memused +=
      (DD_MEM_CHUNK + 1) * sizeof(DdNode);
   mem[0] = (DdNodePtr) unique->memoryList;
   unique->memoryList = mem;
   offset =
      (ptruint) mem & (sizeof(DdNode) - 1);
   mem += (sizeof(DdNode)
      -   offset) / sizeof(DdNodePtr);
      . . .
```

Figure 3. Code containing data structure assumptions

This situation is depicted more abstractly in Figure 1-C. The obstacles associated with the strategy are *encoded*, meaning that they are neither structurally explicit, nor are they implied by comments or conventions. As a result, the participant was unable to use either of the inward or outward reasoning strategies employed by other participants. In the end, the participant simply worked around this difficult code.

3.4 Summary of Results

For all participants, overcoming an obstacle involved significant effort to understand the relevant portions of the crosscutting concern associated with the obstacle. Determining the interface between the broader concern code, and the code related to the change was considered a non-trivial task, especially by the participants who faced implied obstacles and who applied the within strategy. Consistently, participants wanted an answer to the question: If I change this location in the code, how will that crosscutting concern be affected?

4. DISCUSSION

We claim that our paper provides contributions in two areas: empirical evidence of crosscutting concerns and the strategies used in coping with such concerns, and input to future assessment of aspect-oriented programming. In this section, we discuss our contributions in each of these areas.

4.1 Empirical Validity

Our study considered eight separate change tasks. Each task was being performed on a unique system. The systems were implemented in range of programming languages: three systems were implemented in C [12], three in C++ [13], and two in Java [14]. The participants performing the changes were not novice developers: four of the participants were practicing software developers in industry. The questions asked of participants focused on the changes being performed rather than on the crosscutting concerns encountered. Despite the differences in participants, tasks and systems, similarities emerged in the form of the crosscutting code involved, and in the strategies used by the participants to cope with the crosscutting concerns. These similarities increase our confidence that the results are indicative of *real* software developments and that the results may generalize.

Two limitations of our study are the small number of systems and tasks considered, and the short amount of time that we tracked the progress of the developers. Our study had these limitations because it was exploratory in nature. Originally, we had hypothesized that concerns might be more directly linked with change tasks. For instance, a change might correspond with a concern. Through this exploratory study, we found that concerns typically intersected changes. A larger, longitudinal study to further test the hypothesis that concerns intersect change is needed.

4.2 Assessing Aspect-oriented Programming

The results of our study provide a basis for helping to assess aspect-oriented programming. Specifically, if a particular crosscutting concern was modularized, and perhaps separated, we would assume that programmers would not have to choose the around strategy to cope with obstacles encountered when making a change. One could test this hypothesis by taking a system that was used in this study, representing the crosscutting concerns as aspects, subjecting the aspect form of the system to the same change, and then observing the actions of the developer(s). Alternatively, one could follow changes to a system built using aspect-oriented ideas and technology and see if the strategy occurs. We would still expect the change and within strategies to occur as changes were made to an aspect-oriented system. However, we would expect the aspect form of a system would

make it easier for the developers to analyze and understand the interactions between the core change code and the concerns.

5. RELATED WORK

We compare our study to empirical work in two areas: empirical studies of programmers performing software change tasks, and empirical efforts to assess aspect-oriented programming.

Empirical Study of Programmers. A significant amount of work has been undertaken to analyze the cognitive and mental approaches used by programmers to understand source code. Four basic approaches have been characterized: top-down [15, 16], where the programmer begins with understanding of a general nature, bottom-up [17, 18], where programmers begin by reading source code and by mentally forming higher-level abstractions, knowledge-based [19] which involves assimilating domain knowledge and the mental models formed during program analysis, and integrated [20] which incorporates all of the above.

We see all of this work as complementary to our own. These empirical approaches place emphasis on the work practices used and on the types of mental and cognitive models built by programmers while understanding code. Our work looks at a more specific concept: what is the form and role of the code that programmers examine when performing a program change task.

Empirical Work on Aspect-oriented Programming. In a case study on the use of AspectJ to modularize and separate exception detection and handling, Lippert and Lopes noted several strengths and weaknesses of the aspect-oriented approach [21]. In particular, they noted that at certain points when performing a task, programmers needed to see the behavioural effects of aspects on methods of interest. The participants in our study expressed a similar desire: They wanted to see their concern with respect to portions of the code of interest.

Walker and colleagues report on a controlled experiment to investigate whether aspect-oriented programming could ease program maintenance tasks [22]. They reported that programmers found it difficult to reason about a separated concern when the interface between the core code and the concern code was too broad. Restated, the more constrained and defined the interface, the easier it was for programmers to determine the area of influence between the code and concern code. Our study corroborates this result. The narrowest interface occurred when programmers could reason out from their code; when they were able to capture the interface based on information within the core of their task. Participants working in these conditions were able to find relevant portions of the code to change, though they noted that it was a tedious process. The interface in this case was clear: all methods that performed a particular function related to a particular type had to be considered. A wider interface corresponds with the inward-reasoning situation when programmers had to take information from other portions of the code and then had to analyze their core in terms of the assumptions and invariants in the broader code. These participants reported more difficulty in finding those external points of reasoning than those working with outward reasoning. Finally, the widest interface was the one that could not be defined at all, and which lead to the around strategy in which the attempt to understand the concern code was abandoned.

6. SUMMARY

This paper reports on a study conducted to examine where developers encounter crosscutting code during a program change task, and how the developers chose to manage that code. We found that crosscutting code emerged as obstacles that the programmers had to manage when making the desired change.

When obstacle code related to a broader concern was encountered, developers had to try to understand both how the changes they were making affected the crosscutting concern, and how the crosscutting concern affected their change. We discovered they used one of three strategies to deal with the crosscutting concern: in some cases, developers altered the crosscutting code to accommodate the change, in other cases, developers made the change work in the context of the crosscutting code, and in yet other cases, developers worked around the crosscutting code. Each strategy corresponded to a different form of the obstacle code. When there were suitable structural links and a developer could reason out from the obstacle point in the code related to the change to the concern code, the first strategy, the change strategy, was used. When there were behavioural patterns but no structural links, developers reasoned from the concern code into the change code and adopted the second strategy, the within strategy. When neither of these reasoning approaches was possible because of dense and subtle code, developers took the third approach of working around the crosscutting code.

This paper provides empirical evidence to support the existence and type of crosscutting concerns on which aspect-oriented programming approaches are based. This paper also lays the groundwork for further assessment of aspect-oriented programming.

ACKNOWLEDGMENTS

This work was funded, in part by Siemens AG Corporation, in part by a University of British Columbia Graduate Fellowship, and in part by a grant from the National Science and Engineering Research Council of Canada (NSERC).

We thank all participants who provided their time and experiences for our study.

We would also like to thank the anonymous reviewers for their comments on this paper.

REFERENCES

[1] D. L. Parnas, On the Criteria To Be Used in Decomposing System into Modules, *Communications of the ACM,* pp. 1053-1058,1972.

[2] AspectJ™ web site: www.aspectj.org

[3] Hyper/J™ web site: www.research.ibm.com/hyperspace/HyperJ/HyperJ.html

[4] M. Askit, L. Bergmans and S. Vural. An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach, In *Proc of European Conference on Object-Oriented Programming (ECOOP), Lecture Notes in Computer Science* Vol. 615, pp. 372-395, 1992.

[5] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda and C. Lopes, J.-M. Loingtier, and J. Irwin. *Aspect Oriented Programming.* In *Proc. of European Conference on Object-Oriented Programming (ECOOP), Lecture Notes in Computer Science* Vol. 1241, pp. 220-242, 1997.

[6] H. Ossher, M. Kaplan, A. Katz, W. Harrison and V. Kruskal. Specifying subject-oriented composition. *TAPOS*, Vol. 2, No. 3. pp. 179-202, 1996.

[7] P. Tarr, H. Ossher, W. Harrison and S.M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Proc. of the 21st International Conference on Software Engineering*, pp. 107-119, 1999.

[8] C.V. Lopes and G. Kiczales. "Recent Developments in AspectJ™". Aspect-Oriented Programming Workshop, European Conference on Object-Oriented Programming (ECOOP). In *Object-Oriented Technology: ECOOP'98 Workshop Reader*, *Lecture Notes in Computer Science*, Vol. 1543, pp. 398-401, 1998.

[9] C.V. Lopes. *D: A Language Framework for Distributed Computing*, Ph.D. Dissertation, College of Computer Science, Northeastern University, Boston, 1997.

[10] C.V. Lopes and K.J. Lieberherr. Abstracting process-to-function relations in concurrent object-oriented applications. In *Proc. European Conf. on Object-Oriented Programming (ECOOP), Lecture Notes in Computer Science,* Vol. 821, pp. 81-99, 1994.

[11] T. Lethbridge, S. Sim, and J. Singer. Studying Software Engineers: Data Collection Methods for Software Field Studies, Submitted May 2000 to *Empirical Software Engineering*.

[12] B.W. Kernighan and D. M. Ritchie. *The C Programming Language: Second Edition*. Prentice Hall, Englewood, New Jersey, 1988.

[13] B. Stroustrup. *The C++ Programming Language: Second Edition*. AddisonWesley Publishing Co., 1991.

[14] K. Arnold and J. Gosling. *The Java Programming Language*. ACM Press Books, Addison Wesley Longman, 1996.

[15] R. Brookes. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, Vol. 18, pp. 543-554, 1983.

[16] E. Soloway and K. Erlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, SE-10, No. 5, pp. 595-609

[17] B. Schneiderman and R. Mayer. Syntactic/semantic interactions in programmer behaviour: A model and experimental results. *International Journal of Computer and Information Sciences*, Vol. 8 No. 3, pp. 219-238, 1979.

[18] N. Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, Vol. 19, pp. 295-341, 1987.

[19] S. Letovsky. Cognitive Processes in Program Comprehension. In *Empirical Studies of Programmers*, pp. 58-79, 1986.

[20] A. von Mayrhauser, A. Vans. Comprehension processes during large scale maintenance. In *Proc. of the 16th International Conference on Software Engineering*, pp. 39-48, 1994.

[21] M. Lippert and C.V. Lopes. A Study on Exception Detection and Handling Using Aspect-Oriented Programming. In *Proc. 22nd International Conference on Software Engineering*, pp. 418-427, 2000.

[22] R. Walker, E. Baniassad and G. Murphy. An Initial Assessment of Aspect-Oriented Programming. In *Proc. of the 21st International Conference on Software Engineering*, pp. 120-130, 1999.