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: to make it easier to understand, to make it easier to change, and to make it easier to verify. Aspect-oriented programming approaches extend the kind of code that can be modularized. In particular, these approaches provide support for modularizing crosscutting code. We conducted a (mainly inquisitive) study to better understand the kinds of crosscutting code that software developers encounter and to better understand their current approaches to managing this code. In this study we tracked eight participants from industry and academia: each participant was currently evolving a non-trivial software system. We interviewed these participants several times about crosscutting concerns they had encountered and their strategies for dealing with those concerns. We found that crosscutting concerns tended to emerge as obstacles that the developer had to consider to make the desired change. Their choice of strategy to deal with 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, suggest tool support that may be needed to cope or separate crosscutting concerns in existing systems, 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 for many reasons: to make it easier to read, to make it easier to change [Parnas-72], and to make it easier to verify. Aspect-oriented programming approaches [AspJ, HyJ, Ber-92, Kic-97, Oss-96, Tarr-99, Lop-98] extend the kind of code that can be modularized. In particular, these approaches provide support for modularizing crosscutting code.

These approaches were developed based on some kinds of crosscutting code that occur, such as code associated with distribution [Lop-97], synchronization policies [Lop-99] and some kinds of features [Tarr-99]. To our knowledge, no independent empirical studies have been undertaken to consider what kinds of crosscutting concerns software developers working on existing systems would find beneficial to modularize nor how those software developers are currently managing those concerns. 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. Some of these participants were from industry and some were from academia. All were working on non-trivial changes to non-trivial systems.

Analysis of the data collected during the study indicated that each of the developers was forced to consider at least one crosscutting concern. These crosscutting concerns arose from obstacles, such as memory allocation or some unidentifiable functionality, associated with making a change. Each participant reported finding it difficult to deal with this obstacle code. 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 try to cope with the obstacle. The strategy chosen depended on the form of the obstacle code, and on the form of the interaction between the concern code and the core code associated with the change task.

This study and its results make three contributions. First, the results provide empirical evidence about the kinds of crosscutting concerns impacting software developers and the strategies developers are using to cope with these concerns in existing systems. Second, the problems encountered in working with the crosscutting code indicate the need for tool support to work with, and potentially to modularize, this code in existing systems. Moreover, the strategies employed by the developers in coping with the code suggest the kinds of tool support needed. Third, the results provide a basis on which to compare whether the use of aspect-oriented approaches can enable 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 when similar situations where encountered as those described in this paper than 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 mainly inquisitive [Leth-00]. Over a three-week time period, we tracked the progress of participants on a change task to a system that they were involved in 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: some had years of programming experience in an industrial setting; others were graduate students with a range of programming experience. Four of the participants were practicing software engineers from Siemens AG; four were Computer Science graduate students at the University of British Columbia.

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 before the beginning 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 they had not written. Each participant 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 for interviews were prepared in advance. These guidelines were meant to focus the interview, but the specific questions that were asked depended upon the flow of conversation. The participants were not informed of the contents of the interview guidelines in advance. 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 exist in codebases and that developers have to consider. 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 context for, or the meaning of, our questions. Second, there was a danger that the participants who did have some knowledge of this area would jump to responding about popular crosscutting concerns like tracing, debugging, or distribution. Such 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 started to think and 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 names that they might use to describe the kinds of 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 the participant would like to see modularized. Since this question is abstract, the interviewer provided suggestions for 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 we received. Second, we categorized the responses of the participants in terms of how they 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 portions of annotated source code, looking at the form of the statements highlighted. We attempted to spot commonalities in terms of syntax, semantics, or function. The interviewer also examined the code bases of the participants to try to determine whether the changes being made could be characterized as belonging to a particular concern.

3. QUALITATIVE ANALYSIS
Participants commonly 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 problem domain of the application 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.

Participants' initial descriptions of the change task were in terms of easily identifiable structure in the code. Specifically, most participants described the changes 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. The fact that it was easy to describe the change from this perspective was not due to a good modularization of the code; often the code was spread across the code base. Rather, programmers found it easy to identify the code because they could understand the code's purpose and its context within the structure of the application. They could point out portions of the code that corresponded to their change.

A participant described crosscutting code as the target of the change in only one case. This participant was currently working in the area of aspect-oriented programming.

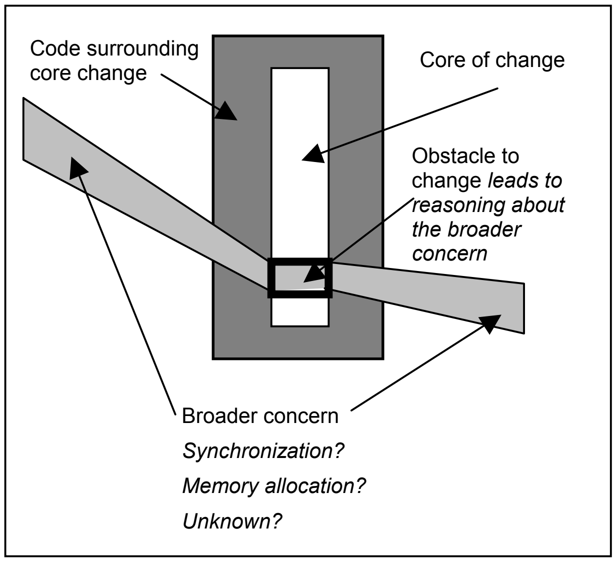
Code surrounding core change
Core of change
Obstacle to change *leads to reasoning about the broader concern*
Broader concern
Synchronization?
Memory allocation?
Unknown?

Figure 1. Obstacles reveal concerns

3.2 Non-straightforward Obstacle Perspective

After participants had generally described their change task, and after they had pointed out the locations in the code that 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 while making the change.

Figure 1 provides an abstract representation of the experiences of the participants. As long as the change was within a structural context, the participants could understand and conceptualize the change. The white vertical rectangles in Figure 1 represent the core change code that was associated with structure. However, as the change was being made, the programmers tended to encounter obstacles (shown in black). These 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, one participant wanted to change the way in which a user interface was distributed in a distributed system. This change involved modifying and testing the user interface sections, and it also involved testing to ensure that the distribution itself had not broken down. Another participant was adding a feature to the system; this participant had to ensure that the change was consistent with other similar changes, and that it did not damage existing functionality. To make the change, the participant had to overcome the obstacle(s) and to try to understand the entire underlying concern (shown in light grey in Figure 1) 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.

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	Computation assumptions built into data structures. Undecipherable obstacle portions	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 Resource speed	Change
8	Adding cancellation notification to an existing system	Multithreading Behavioural consistency	Within

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, or the locations at which the original 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 2-A depicts this situation. These obstacle points 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. This situation was particularly true for participant two. For this participant, the obstacle points, or points of intersection 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 2-B, the obstacle points were implied. Comments alerted both 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 entire code base in order to work within the obstacles. 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 3 shows the inward reasoning, and resultant code, used by participant one. This participant's task was to separate operating systems pre-fetching code into a separated aspect-like module [Coa-01]. Boxes A1 and B1 refer to code identified as belonging to the broader synchronization concern. As was true of both participants' code, there were no clear intersection points in the core code with the obstacle code. Hence, no obstacle points are visible in Figure 3. Based on previous knowledge, and on comments not shown in the figure, participant one reasoned about the concern code in boxes A1 and B1 in addition to reasoning about the rest of the synchronization conventions for the system. From the broader concern code, participant one reasoned inward about the pre-fetching code that formed the core of the change task. To work within the synchronization conventions, participant one had to add, as part of the change, portions of code related to the synchronization concern. This code is shown in boxes A2 and B2; strictly speaking, this code was not directly related to the core of the change. The inclusion of this code ensured that the locking invariants encoded in the synchronization concern were maintained. The reasoning from boxes A1 and B1 inward to the core change thus resulted in the addition of code to the re-modularized core.

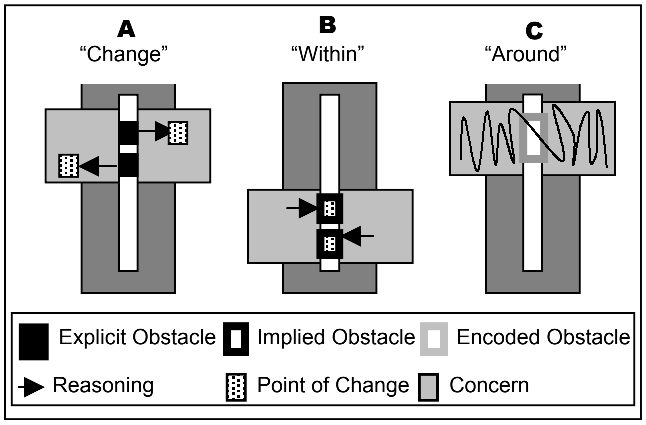

Figure 2. Obstacle types: Core-Concern Intersections

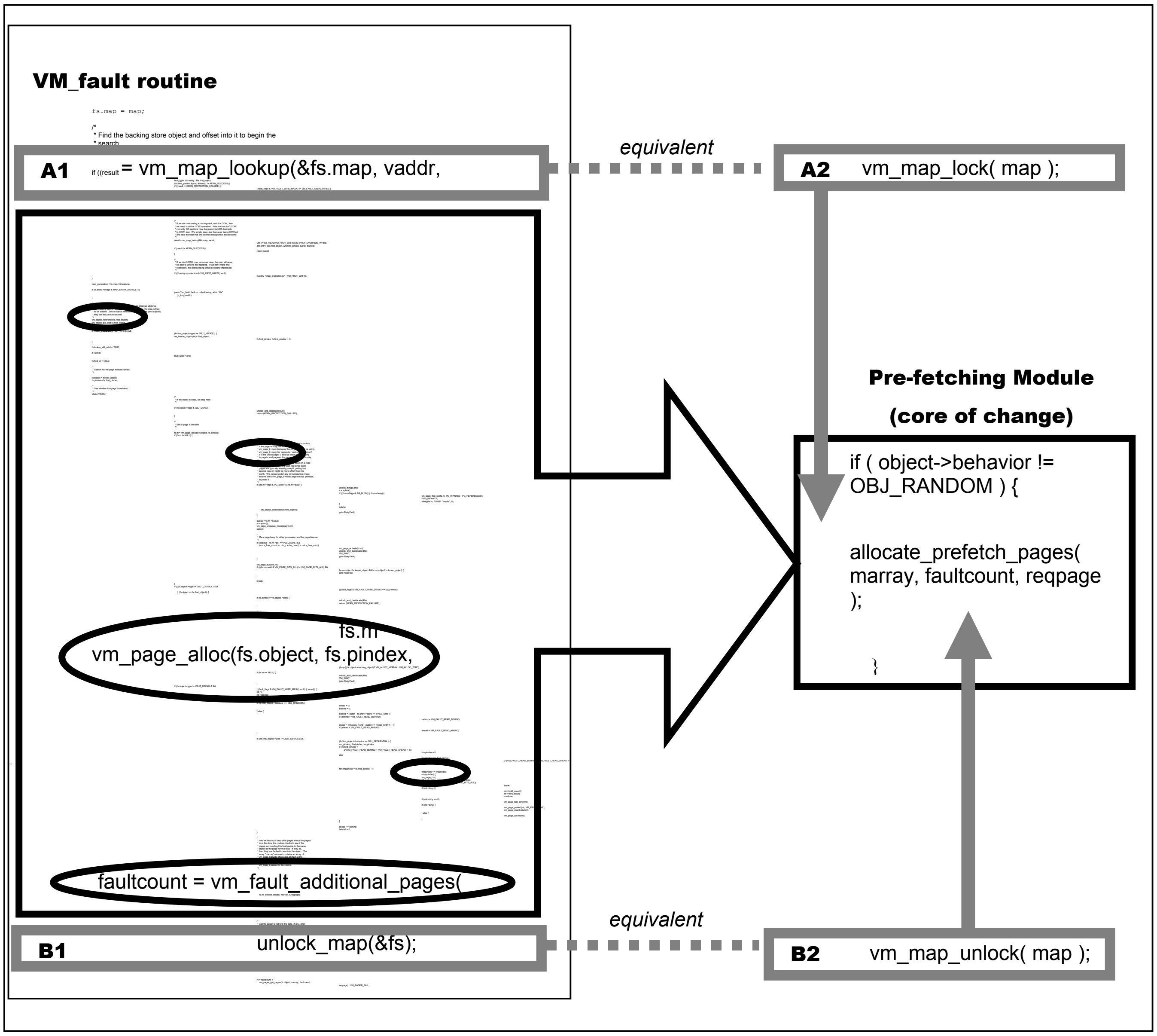
VM_fault routine
fs.map = map;
A1
if ((result = vm_map_lookup(&fs.map, vaddr,
equivalent
A2
vm_map_lock(map);
vm_page_alloc(fs.object, fs.pindex,
faultcount = vm_fault_additional_pages(
Pre-fetching Module
(core of change)
if (object->behavior != OBJ_RANDOM) {
allocate_prefetch_pages(marray, faultcount, reqpage);
}
equivalent
B1
unlock_map(&fs);
B2
vm_map_unlock(map);

Figure 3: Code alterations show inward reasoning

In both cases, participants were unable to cleanly determine when they had covered all of the code related to their change. Our examination of their code yielded limited similarities about the nature of the external concern code. 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 the obstacle code and the concern it pointed to. In this code, it is unclear why particular data structures are altered in particular ways, and it is unclear why the ordering of certain computations is important. For instance, the obstacle code encountered by participant four assumed that a data structure of a certain number of bytes would be used. This number was hard-coded as an assumed constant throughout this piece and other pieces of code, but it was not structurally explicit in this piece of code where the number was being assumed. For instance, parts of the algorithmic code assumed this constant but did not refer to it.

This situation is depicted in Figure 2-C, which shows obstacles associated with this 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. When asked, the participants described that even if they had been given a view of the crosscutting concern, it would likely still have required significant reasoning on their part to decipher the effect of their actions on the concern code. 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 met with 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 three areas: empirical evidence of crosscutting concerns and the strategies used in coping with such concerns, directions for tool support to help manage crosscutting concerns in existing systems, and input to future assessment of aspect-oriented programming. In this section, we discuss our contributions in each of these areas.

4.1 Study 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 [Ker-88], three in C++ [Str-91], and two in Java [Gos-96],. 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 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. The presence of these similarities in the context of the differences between participants, systems, and tasks increases 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. A longitudinal study of more systems that, in particular, subjects the strategies we discovered to further scrutiny is likely warranted.

4.2 Tool Support for Obstacles

Based on our analyses, the tool support that would likely have been useful to our participants to cope with emerging obstacles varies with the three categories of approaches they employed.

Those participants who worked with obstacles in a change strategy may have been able to effectively deal with the concerns if they could have identified and reasoned about the concern code as a block. Tools for finding latent aspects, such as the AspectBrowser [Gris-01], AMT [Hann-01], and FEAT [Mrob-01] may be sufficient for these obstacles since there were lexical and structural clues in the source to identify the obstacle code.

The participants who worked with obstacle code in a within strategy may have benefited from tools that provide views on parts of crosscutting concerns, and from tools that provide more extensive analysis of the source than is found in the existing set of aspect finding tools. For instance, participant one did not need to understand the entire synchronization scheme for the system. This participant needed a view on the relevant parts of the synchronization scheme pertaining to the code being changed. A means of determining the pertinent parts of the code through searches on patterns of operations and then being able to view those results in the context of the change code may have been helpful.

The participants who used the around strategy may have benefited from measures that could have told them the situation was largely hopeless before they had invested a significant amount of time. Such measures would need to bring out the subtlety of the code: simple metrics based only on structural coupling would not likely suffice.

4.3 Assessing Aspect-oriented Programming

The results of our study provide a basis for helping to assess aspect-oriented programming. Specifically, we would assume that if crosscutting concerns were modularized, and perhaps separated, that programmers should not have to choose the around strategy to cope with obstacles encountered when making a change. One could test this hypothesis either by taking a system that was used in this study, representing the obstacle code as aspects, and then subjecting the aspect form of the system to the same change and observe 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 change code and the obstacles.

5. RELATED WORK

We describe the relation of our study to empirical work in two areas: empirical studies of programmers performing parts of 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. Broadly, four approaches have been characterized: top-down [Brooks-83, Sol-84], where the programmer begins with understanding of a general nature, bottom-up [Schn-79, Penn-87], where programmers begin by reading source code and by mentally forming higher-level abstractions, knowledge-based [Let-86] which involves assimilating domain knowledge and the mental models formed during program analysis, and integrated [vonM-94]

Storey and colleagues [Sto-99] describe a set of cognitive issues to be considered when designing a software exploration tool. They examined a number of cognitive models of program comprehension, and gave examples of how the use of these models could be facilitated by tool support.

Singer and Lethbridge [Sing-98] conducted a set of field studies in which they analyzed the work practices of developers as a means of directing tool development. They collected information in four ways: through tool usage statistics, through a web questionnaire, through group studies, and by shadowing a developer for an extended period of time. Based on this data, they concluded that programmers need support when searching code bases, in terms of being able to store the results of their search, and that they need support for recalling their movement through code bases.

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 [Lipp-00]. In particular, they noted that at certain points in performing tasks, programmers needed to see the behavioural effects of aspects on methods of interest. This finding is similar to one of the results of this study: programmers want to see their concern with respect to portions of the code of interest. The similarity in these findings reinforces the result that programmers need to see the effects of the relevant portions of a broader concern on particular portions of code.

Walker and colleagues report on a controlled experiment to investigate whether aspect-oriented programming could ease program maintenance tasks [Wal-99]. 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 relates to 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. These different strategies corresponded to different forms 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. The strategies for coping with crosscutting concerns in existing systems suggest particular tool support that is needed. As one example, developers may benefit from views that can be computed on demand to show the ramifications of a change on a particular set of crosscutting concerns. Finally, this paper 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.

A shorter, earlier version of this work was reported in the Workshop on the Advanced Separation of Concerns held at Object-Oriented Programming, Languages and Applications 2000. (www.cs.ubc.ca/labs/spl/papers/2000/oopsla2000-asoc-eb.html)

REFERENCES

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

[Ber-92] L. Bergmans, M. Askit, K. Wakaita and A. Yonezawa. An Object-Oriented model for extensible concurrent systems: The composition-filters approach. Technical Report, 1992

[Brooks-83] R. Brookes. *Towards a theory of the comprehension of computer programs*. International Journal of Man-Machine Studies, 18:543-554, 1983

[Coa-01] Yvonne Coady, Gregor Kiczales, Mike Feeley and Greg Smolyn. *Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code, FSE 2001*.

[Gos-96] K. Arnold, J. Gosling*: The Java Programming Language*. ACM Press Books, Addison Wesley Longman, 1996.

[Gris-01] W. G. Griswold, *Coping with Software Change using Information Transparency.* Technical Report CS98-585, Department of Computer Science and Engineering, University of California, San Diego, April 1998 (revised August 1998). A version of this paper is to appear at Reflection 2001, Kyoto, 2001.

[Hann-01] Hannemann, J., and Kiczales, G. *Overcoming the Prevalent Decompositionof Legacy Code*, Proceedings for the Workshop on Advances Separation of Concerns at the International Conference on Software Engineering, 2001

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

[Ker-88] Kernighan, B. W., and Ritchie, D. M. *The C Programming Language*. Prentice Hall, Englewood, New Jersey, 1988. The second edition

[Ker-99] M. Kersten and G. Murphy, *Atlas: A Case Study in Building a Web-based Learning Environment using Aspect-Oriented Programming*. In Proceedings of OOPSLA'99. Denver, CO, USA. November 1999, ACM Press, pp. 340-352, 1999.

[Kic-97] 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, LCNS 1241, pages 220-242, 1997.

[Let-96] S. Letovsky *Cognitive Processes in Program Comprehension.* In Empirical Studies of Programmers, 58-79. Ablex Publishing Corporation, 1986.

[Leth-00] Lethbridge, T., Sim, S. and Singer, J. (1998 July), *Studying Software Engineers: Data Collection Methods for Software Field Studies*, Submitted May 2000 to Empirical Software Engineering

[Lipp-00] Martin Lippert and Cristina Videira Lopes. *A Study on Exception Detection and Handling Using AspectOriented Programming*. Proc. 22 nd International Conference on Software Engineering, 2000.

[Lop-97] Lopes C. V, "*D: A Language Framework for Distributed Computing*", Ph.D. Dissertation, College of Computer Science, Northeastern University, Boston, 1997

[Lop-98] Christina Vedeira Lopes and Gregor Kiczales. "*Recent Developments in AspectJ™*". Aspect-Oriented Programming Workshop, ECOOP'98. In Object-Oriented Technology: ECOOP'98 Workshop Reader, S. Demeyer, J. Bosch (eds), Lecture Notes in Computer Science, Vol. 1543, pp.398-401, Springer, 1998.

[Lop-99] Lopes, **C**.V., Lieberherr, K.J., 1994. *Abstracting process-to-function relations in concurrent object-oriented applications*. Proc. European Conf. on Object-Oriented Programming (ECOOP'94), LNCS 821, Springer-Verlag, 81--99

[MRob-01] Martin P. Robillard, Gail C. Murphy *Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies.* Technical Report UBC-CS-TR-2001-13, Department of Computer Science, University of British Columbia, 10 September 2001.

[Ols-87] G. M. Olson, S. Sheppard and E. Soloway *Change Episodes in Coding: When and How Do Programmers Change Their Code*. Empirical Studies of Programmers: Second Workshop.. 1987: Norwood, NJ, Ablex. pp. 185-197.

[Oss-96] H. Ossher, M. Kaplan, A. Katz, W. Harrison, and V. Kruskal. *Specifying subject-oriented composition*. TAPOS, 2(3):179-202, 1996.

[Parnas-72] D. L. Parnas, "*On the Criteria To Be Used in Decomposing System into Modules*" Communications of the ACM pp. 1053-1058 (December 1972)

[Penn-87] N. Pennington. *Stimulus structures and mental representations in expert comprehension of computer programs.* Cognitive Psychology, 19:295-341, 1987.

[Sch-79] B. Schneiderman, R. Mayer. *Syntactic/semantic interactions in programmer behaviour: A model and experimental results*. International Journal of Computer and Information Sciences, 8(3):219-238, 1979.

[Sing-98] J. Singer and T. Lethbridge. *Studying work practices to assist tool design in software engineering*. In 6th International Workshop on Program Comprehension (WPC'98), Ischia, Italy, pages 173--179, June 1998.

[Sol-84] E. Soloway, K. Erlich. *Empirical studies of programming knowledge*. IEEE Transactions on Software Engineering, SE-10(5):595-609

[Sto-99] M.-A.D. Storey, F.D. Fracchia and H. A. Müller. *Cognitive Design Elements to support the Construction of a Mental Model During Software Exploration*. Journal of Software Systems, special issue on Program Comprehension, volume 44, pp. 171-185, 1999.

[Str-91] Bjarne Stroustrup. *The C++ Programming Language: Second Edition*. AddisonWesley Publishing Co., Reading, Mass., 1991.

[Tarr-99] Peri Tarr, Harold Ossher, William Harrison and Stanley M. Sutton. *N degrees of separation: Multi-dimensional separation of concerns.* In Proceedings of the 21st International Conference on Software Engineering, pages 107-119, May 1999.

[vonM-94] A. von Mayrhayser, A. Vans. *Comprehension processes during large scale maintenance*. In Proceedings of the 16th International Conference on Software Engineering, pages 39-48, 1994.

[Wal-99] R. Walker, E. Baniassad and G. Murphy. *An Initial Assessment of Aspect-Oriented Programming.* In Proceedings of the 21st International Conference on Software Engineering, pages 120-130, May 1999.