Where are Programmers Faced with Concerns?

Elisa L.A. Baniassad and Gail C. Murphy
Department of Computer Science
University of British Columbia
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

This paper describes the results of a small exploratory study performed to elucidate crosscutting concerns in existing codebases. Our hypothesis was that that the code directly associated with program change tasks would be representative of crosscutting concerns. To our surprise, we found that it was more often obstacles in the code encountered while making a change that indicated concerns.

1. Introduction

Several new mechanisms have been developed recently to improve the support for separating crosscutting concerns in code, including AspectJ™[1][10], Hyper/J™[2][8], and composition filters[3]. In the literature, many potential kinds of concerns suitable for separation have been suggested, including synchronization policies [11], exception handling [9], security [4], and features [8]. To date, a few case studies [1][9] and some experiments[12] have been conducted to investigate the impact of separating concerns. However, little is know about the kinds of concerns practicing software developers would find beneficial to separate to aid common development tasks. Improving our understanding of the concerns of interest to developers may help direct the development of tool support to aid the separation of concerns.

To address this issue, we have conducted a small exploratory study to determine the kinds of concerns programmers would find useful to separate in existing codebases. This study involved developers in both an academic and an industry setting.

The hypothesis of our study was that the portions of code related to a program change task would correspond to a crosscutting concern, and thus that tracking program change tasks would help identify concerns in a codebase.

The study involved eight participants who were either currently, or had recently been, involved in changing an existing codebase. Each participant was interviewed three times. In each interview, a participant was asked a series of questions about the nature of their change task, and their approach to the task.

Analysis of the data collected during the study indicated that our hypothesis did hold: we could identify crosscutting concerns from program change tasks. However, we did not identify the concerns in the way we had anticipated. Instead of crosscutting concerns being directly associated with the change at hand, such concerns more often arose from obstacles associated with performing the change. Participants related that everything was going fine with the change task until they ran into an obstacle, such as a piece of the change which affected memory allocation, or synchronization, or some sort of unidentifiable functionality in the system. All of the participants found it difficult to deal with obstacle code because it required them to address an embedded crosscutting concern.

2. Description of the Study

Background of the participants

Because this was an exploratory study, we chose participants from 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. The study involved eight participants in total: four engineers from Siemens AG and four graduate students at the University of British Columbia.

Only two of the participants were familiar with the concept of separation of concerns and the newly developed mechanisms available to support separation. One of these two participants was actively applying aspect-oriented programming [6] ideas in the change task examined. Two of the other participants had never heard of the separation of concerns concept until the study

To participate in the study, we required that a participant be working on, or recently have worked on, a program change task on source code they themselves had not written.

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.

General Format

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

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. During each interview, we wanted to determine four different pieces of information:

- the program change task of the participant,
- the approach of the participant to the task,
- how the participant figured out what pieces of code needed to change, and
- whether the participant thought the change was difficult to make and if so, why it was difficult.

To help focus the discussion, participants were asked to identify which portions of code had, and 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.

How the questions were posed

The purpose of the study was to look for crosscutting concerns in existing codebases. Rather than ask the participants directly what concerns they wished to see separated, we asked them questions about the change task on which they were working, and attempted to glean concerns from their responses.

There were three reasons for taking this approach. First, most of the participants had never thought about separation of concerns. When we attempted to pose questions that directly asked about concerns, the participants were unable to understand the context for, or meaning of, our questions. Second, there was a danger that the participants who did have some knowledge of separation of concerns would jump to responding about 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 readily answer from their own experience, and then analyzing their responses 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. By the end of an interview, participants typically 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 they might use to describe the kinds of code they were examining, or methods 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 separately. Since this question is vague, the interviewer provided suggestions for answers. An example answer might have been "all the code pertaining to the database system" or "all the code related to printing".

Method of Qualitative Analysis

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

Our examination involved three passes of the transcripts. First, we looked over 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 portions of the annotated source code, attempting to spot commonalities in terms of syntax, semantics, or function. The interviewer examined the participants' codebases 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…" they reached an obstacle.

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 itself 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. For instance "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 encapsulation of the code; often the code was spread across the codebase. 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 were able to point out portions of the code that corresponded to their straightforward changes.

In only one case did a participant describe code pertaining to a crosscutting concern in the source as the target of the change. This participant was currently working in the area of aspect-oriented programming.

Non-straightforward Obstacle Perspective

After participants had described the general concepts of the change upon which they were working, and after they had pointed out the locations in the code which 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 they had encountered while making the change.

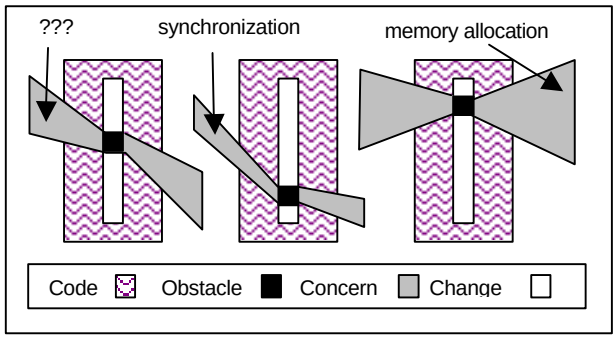

Figure 1. Obstacles reveal concerns

Figure 1 provides an abstract representation of the programmers' experiences. As long as the change was within a structural context the programmers could understand and conceptualize the change. The white inner vertical rectangles in Figure 1 represent the code associated with structure that needed to change. However, as the change was being made, the programmers tended to encounter obstacles (shown in black). These obstacles comprised portions of code which were relevant to the task but that also affected an underlying concern. For example, one participant wanted to change the way user interface information was passed around in a distributed system. As would be expected, this change involved testing the user interface code after the change. However, in addition, it also involved testing that the distribution in the program was still working after the change. The distribution code, in this case, was the obstacle because the programmer had to try to understand and test the entire underlying concern (shown in light grey in Figure 1) that led to the presence of that portion of code in order to make the change. Because that underlying concern code was neither well modularized nor well documented, it was difficult to conceptualize and reason about.

Table 1 shows the program change tasks for each participant, along with the obstacles encountered.

Table 1: Participants' task descriptions

Participant	Straightforward Structural view	Non-straightforward Obstacle View
1	Moving particular computation to an aspect-like module.	• Synchronization • Performance
2	Changing table representation	• Memory Allocation
3	Changing matrix calculation	• Memory Allocation
4	Tailoring a matching algorithm for a specific purpose	• Computation assumptions built into data structures. • Undecipherable obstacle portions
5	Changing packaging of user interface mechanism	• Distribution • Tracing
6	Changing the mathematical model applied	• Security issues • Communication protocols • Hardware platform dependencies.
7	Changing printing look and feel	• User Interface consistency • Resource speed
8	Adding cancellation notification to an existing system	• Multithreading • Behavioural consistency

When faced with an obstacle, the programmers chose one of the following three strategies:

1. Alter the relevant "obstacle" code to enable the change task.
2. Understand, but not change, the underlying concern associated with the obstacle sufficiently to make the change work within it.

3. Completely alter the change task itself to account for the obstacle without understanding the obstacle.

Participants two, three and four chose option three. They significantly rethought their approach to the change so as to avoid dealing with, or understanding the obstacle code. In these cases, the participants decided that they could not adequately understand or address the obstacle. For instance, Participant four ran into memory allocation problems after making what should have been a simple change to a table representation. Rather than attempting 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 a change had not been made.

Participants one, five, six and eight chose option two. They worked hard to understand the affect of their code on the crosscutting concern that presented an obstacle to their change, and worked within the conventions of the concern. Participant eight had to perform considerable testing to ensure the obstacle had been dealt with appropriately.

Participant seven was the only one to change the relevant portions of the crosscutting concern to suit the change. This approach was facilitated by the fact that the changes were at the user interface level, and thus it was easier to test the effect of the changes.

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 were given a view of the crosscutting concern, it would still require significant reasoning on their part to understand how their change impacted and relied upon the concern. This observation seems to indicate the need for a ramification-based or context-sensitive view of the crosscutting concern with relation to a particular portion of code. Such a view would allow programmers to ask the question "if I change this location in the code, how will that crosscutting concern be affected?".

To provide some evidence of our conclusions, we present the cases of two of the eight participants in more detail.

Participant One. The change task of Participant one involved moving source code related to a certain crosscutting computation into an aspect-like module. When we first began our discussions, the participant was able to clearly state the kind of code that was the target of the change, and was also able to point out the locations of such code in the source. In the participant's original description, the change seemed simple to conceptualize: a certain kind of code was being moved. This description exemplifies what we have termed a straightforward structural perspective of the problem.

We then probed further and asked if the change was as straightforward as it sounded. It was at this point that the obstacle perspective was revealed. In fact, the change was riddled with complexity. Participant one had to keep in mind that once code had moved to its new location, it had to be realigned with the synchronization policy for the entire application. In addition, the participant had to take into account the performance impact of the change. It was not until after the participant began changing the code that the *extent* of the impact became evident.

Participant one's interest in the synchronization crosscutting concern did not extend across the entire system. Instead, the participant's desire was simply to understand the ramifications of the crosscutting concern on the moved code, and vice-versa. The participant was thus interested in a local context-sensitive view of the crosscutting concern.

Participant Seven. Participant seven wanted to make a simple change to the printed output of his tool. He wanted the entire content of scrollable windows to print, rather than simply what was showing on the current screen. The participant was able to easily show the interviewer the places in the source that needed to be touched to make this change. The participant was able to describe the change simply, in terms of the relevant modules in the source.

We then started the second phase of our questioning. What problems have you encountered while trying to make this change? The participant noted that one of the requirements set out by the client was that the printed documents and the user interface should be identical in look-and-feel. This constraint meant the participant was also forced to examine all the code related to the screen version of the user interface. This activity had not been in the participant's initial assessment of the scope of the task.

Of course, since only certain portions of the screen GUI code were relevant, the participant first had to determine how to make minimal changes without destroying the design of the system with a series of "hacks". This points again to the need for context-sensitive view of the crosscutting concern.

4. Summary

This position paper reports on a small, exploratory study conducted to examine which concerns a programmer might find useful to separate to help support program development tasks. The study focused on tracking program change tasks to identify concerns in existing source codebases.

Our study shows that code pertaining directly to the program change tasks does not always indicate a crosscutting concern. However, the obstacles encountered by programmers when trying to make a change do often indicate crosscutting concerns.

We also found that when programmers encounter crosscutting obstacles in their code, they need to understand both how their localized change affects the crosscutting concern, and how the crosscutting concern affects the change they are trying to make. In general, the programmers did not think it would be necessary to see entire crosscutting concerns separately from the rest of the source, but rather they might find it helpful to be able to see the portions of the concern affected by the change.

Our small study shows that the kind of separated views of source code useful to developers depends upon the tasks that they are performing. In particular, when performing maintenance tasks, developers may benefit from local, context-sensitive views of a crosscutting concern.

Acknowledgements

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

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

References

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

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

[3] M. Askit, L. Bergmans and S. Vural. *An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach*, Proceedings ECOOP'92, LNCS 615, Springer, June 1992, pp. 372-395. File: ftp://ftp.cs.utwente.nl/pub/doc/TRESE/LanguageDbase.ps.Z

[4] R. E. Filman, *Injecting Ilities*. Workshop on Aspect-Oriented Programming, ICSE-20, Kyoto, Japan, April 1998

[5] 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.

[6] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. *Aspect Oriented Programming*. In Proceedings of ECOOP'97, LCNS 1241, pp. 220-242, Springer, 1997.

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

[8] 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, pp. 107-119, May 1999.

[9] M. Lippert and C. V. Lopes. *A Study on Exception Detection and Handling Using Aspect-Oriented Programming.* Technical Report P9910229, Xerox PARC, Number CSL-99-1, December 1999.

[10] Christina Vedeira Lopes and Gregor Kiczales. *Recent Developments in AspectJä*. Aspect-Oriented Programming Workshop, ECOOP'98. In Object-Oriented Technology: ECOOP'98 Workshop Reader, S. Demeyer, J. Bosch (eds), LNCS 1543, pp.398-401, Springer, 1998.

[11] C. V. Lopes and K. J. Lieberherr. *Abstracting Process-To-Function Relations in Concurrent Object-Oriented Applications.* In Proceedings of ECOOP'94, Springer-Verlag, pp. 81-99, 1994

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