A Study of Program Evolution Involving Scattered Concerns

Martin P. Robillard and Gail C. Murphy

Department of Computer Science
University of British Columbia
2366 Main Mall, Vancouver, BC
Canada V6T 1Z4

{mrobilla,murphy}@cs.ubc.ca

Technical Report UBC-CS-TR-2003-06
March 26, 2003

ABSTRACT

Before making a change to a system, software developers typically explore the source code to find and understand the subset relevant to their task. Software changes often involve code addressing different conceptually-related segments of the implementation (*concerns*), which can be scattered across multiple modules. These scattered concerns make it difficult to reason about the code relevant to a change. We carried out a study to investigate how developers discover and manage scattered concerns during a software evolution task, and the role that structural queries play during the investigation. The task we studied consists of a developer adding a feature to a 65kloc Java code base. The study involved eight subjects: four who were not briefed about scattered concerns, and four who were trained to use a concern-oriented investigation tool. Analysis of the navigation among the different program elements examined by the subjects during the task shows evidence that, independent of whether concern-oriented tool support was available, the most successful subjects focused on specific concerns when planning their task, and used a high proportion of structural queries to investigate the code. In addition to the study results, this paper introduces two novel analyses: *navigation graphs*, which support the analysis of a subject's behavior when investigating source code, and *variant analysis*, which is used for evaluating the results of a program evolution task.

1. INTRODUCTION

Before performing a modification to a software system, developers typically explore the system's source code to find and understand the subset relevant for the task. In all but the simplest of cases, it is unrealistic to expect developers to master the complete details of a system's design and implementation prior to undertaking a modification. Rather, a developer must uncover the knowledge necessary to perform the task adequately and efficiently. When the subset of the code relevant to a change is encapsulated within a module, the modification task is usually straightforward. However, few modifications have this property. More often, the code relevant to a change is scattered across many modules [14], and involves different *concerns*, or segments of the implementation of a system which are conceptually related. This scattering of concern code sometimes occurs because the original system decomposition did not encapsulate important design decisions [10]. Other times it occurs because software evolution tasks involve *emerging* concerns: design and implementation decisions that were not encapsulated in a module during development, but do need to be considered as a unit for the purpose of an evolution task. The ever-changing nature of software [11] makes emerging concerns practically inevitable.

For many years now, software development environments have included support for eliciting structural links in source code as a means of helping developers discover and reason about interactions between scattered pieces of a program. As an early example, the Interlisp-D environment provided cross-referencing support [13]. This support continues in today's development environments, such as the Eclipse platform [9]. However, these basic cross-referencing facilities provide little help for managing and understanding the information discovered.

To address these shortcomings, specialized program navigation tools have been proposed to account for scattered concerns. Conceptual Modules [1] allow a developer to form logical modules composed of scattered lines of code, and support an investigation of the control- and data-flow relationships between the different logical units. The Aspect Browser [6] allows users to find and assess concerns based on which lines of code match user-specified regular expressions. The JQuery tool [7] allows a developer to form specialized browsers to navigate code, and allows the developer to perform queries in these browsers while retaining navigation context. Our Feature Exploration and Analysis tool (FEAT) allows developers to capture concerns in Java source code in terms of program elements and structural relations [12]. All of these tools assume that concern-oriented program investigation is a possible and effective way to address a software evolution task. Unfortunately, we have found little empirical work that validates the link between concern-oriented program investigation and success at software evolution tasks.

To bridge this gap, we have carried out a study to investigate how developers manage scattered concerns during a software evolution task, and the role that structural queries play in the investigation of scattered concerns. A case in our study consisted in a developer adding a feature to a medium-sized code base.[1] To investigate a variety of program evolution strategies, we replicated this case with eight subjects. Four subjects were not informed about the possibility of viewing the task in terms of concerns. Four other subjects were trained to use our FEAT concern investigation tool. A case was divided into a program investigation phase and a program change phase. We analyzed the sequence of program elements examined by the subject during both phases, and constructed graphs of the program navigation each subject performed to carry out the task. We have also analyzed each solution to the program evolution task, the notes used by the subjects of the study, and comments the subjects provided during interviews. These analyses provide evidence that, independent of whether concern-oriented tool support was available, the successful subjects focused on specific concerns while planning their task. We have also observed that queries about structural relations between program elements, such as method calls, played an important role in planning the changes in successful cases.

This paper provides three contributions. First, we show evidence to support two current assumptions in the literature: *a)* concern-oriented program investigation is associated with success at program evolution tasks, and *b)* structural queries are an important component of successful concern-oriented program evolution. The other two contributions relate to the analysis of the data. We present a novel technique: the construction of a *navigation graph*, which summarizes the sequence of program elements a developer examines in the source code. Analyses of such graphs can provide insight into the behavior of programmers during program evolution tasks. We also present another novel technique, *variant analysis*, which produces a partial ordering of the quality of the results of an evolution task based on the variants in the implementation of different components of the task.

In section 2, we state our formal research hypotheses. In section 3, we describe the methods we used in our study. In section 4, we present our results, and in section 5, we report on our evaluation of the research hypotheses, and discuss the threats to the validity of our study. Section 6 discusses other studies of programmers addressing software evolution tasks. Finally, section 7 summarizes and concludes the paper.

2. RESEARCH HYPOTHESES

Previous empirical studies have shown that, at least for some software evolution tasks, scattered concerns pose additional challenges to programmers [2, 8], and that a concern-oriented approach to finding and managing code might be helpful [1, 6, 12, 18]. However, this prior work makes two assumptions:

1. Software developers can reason effectively about the design and implementation of a system in terms of a collection of scattered concerns.
2. Software developers can determine what concerns may be associated with a change task.

Furthermore, as we mentioned earlier, there is a tacit assumption in many tools, including our own [12], that structural information in source code, such as method calls, can help developers find and reason about interactions between different, scattered pieces of a program.

To test these assumptions, we posit the following two hypotheses.

H1 *A systematic investigation of the code relevant to a change, addressing a single concern at a time, leads to a more successful evolution task.*

H2 *Structural queries help a software developer complete a concern-oriented software evolution task successfully.*

The justification for H1 is that the cognitive effort required to understand a focused set of design and implementation decisions addressing a single concern is smaller than the effort required to understand all of the code relevant to a change task at once.

The justification for H2 is that navigating between two program elements along a structural relation (as opposed to lexical similarity, file locality, or others) reinforces the understanding of how different, scattered elements fit together in the implementation of a concern.

3. RESEARCH METHODS

3.1 Study design and motivation

To test the hypotheses, we needed to use a realistic task and a system whose size precluded any systematic understanding of the entire code base by the subjects. To meet these constraints, we chose a 3-hour task on a 65kloc code base for a text editor. Including training and preparation, the time for a case was between 4 and 6 hours. Because of the difficulty of finding experienced software developers with this much time to devote to an experimental task, and because of the number of variables that affect the performance of a task, we opted for an evaluation of the hypotheses based on analytical generalization [20], as opposed to inferential statistics. We summarize the logic of analytic generalization, a technique generally used to evaluate case studies, in section 5.1.

To make our main independent variable (the level of concern-orientation) vary between cases we divided the subjects into two groups: a control group and a FEAT group. Subjects assigned to the control group were not given any specific background about approaching a software evolution task by analyzing concerns. Subjects assigned to the FEAT group were given training with the FEAT tool, which supports finding and analyzing concerns in source code based on structural relations between program elements.

3.2 The Task

The target system for our task is the jEdit text editor (version 4.6-pre6).[2] jEdit is written in Java and consists of 64 994

[1] We used the term *case*, as opposed to *trial* or *sample*, because it is more consistent with the nature of our analysis.

[2] `http://www.jedit.org`

non-comment, non-blank lines of source code, distributed over 301 classes in 20 packages. Among other features, jEdit saves open file buffers automatically. Our case focuses on this autosave feature.

In version 4.6-pre6, any changed and unsaved (or dirty) file buffer is saved in a special backup file at regular intervals (e.g., every 30 seconds). This frequency can be set by the user through an Options page brought up with a menu command in the application's menu bar. If jEdit crashes with unsaved buffers, the next time it is executed, it will attempt to recover the unsaved files from the autosave backups. A user can disable the autosave feature by specifying the autosave frequency as zero. This option is undocumented, and can only be discovered by inspecting the source code.

Our task consisted of the following modification request.

Modify the application so that the users can explicitly disable the autosave feature. The modified version should meet the following requirements.

1. *jEdit shall have a check box labeled "Enable Autosave" above the autosave frequency field in the Loading and Saving pane of the global options. This check box shall control whether the autosave feature is enabled or not.*
2. *The state of the autosave feature should persist between different executions of the tool.*
3. *When the autosave feature is disabled, all autosave backup files for existing buffers shall be immediately deleted from disk.*
4. *When the autosave feature is enabled, all dirty buffers should be saved within the specified autosave frequency.*
5. *When the autosave feature is disabled, the tool should never attempt to recover from an autosave backup, if for some reason an autosave backup is present. In this case the autosave backup should be left as is.*

This modification request requires understanding different implementation concerns: each concern involves code scattered in at least two locations. For example, one concern is how properties are displayed and managed by the user interface. Another is how the autosave timer works. Understanding the complete set of functionality related to the change task involves reasoning about the use of approximately 5 fields and 27 methods scattered in 10 classes. The change, as initially performed by an author of this paper in preparation for the study, amounted to about 65 lines scattered in 6 classes.

3.3 Subject Selection

Subjects were recruited through a mailing list for the Department of Computer Science at the University of British Columbia, and through personal contacts. Subjects were required to have Java programming experience, and experience with the maintenance of medium-to-large systems. Student applicants with programming experience gained through cooperative work terms and graduate research projects were accepted for the study, since this level of experience corresponds to the one of entry-level professional developers. No current member of our research group was accepted for this study. Subjects were paid for their time at an hourly rate of 20 CND$. An initial set of subjects was randomly assigned to the two different groups according to two blocking variables: status (professionals versus students), and prior experience with the integrated development environment used for the study (see below). Some adjustments to the initial assignment were required as some subjects dropped out or were disqualified. To avoid bias, subject replacement was done by taking the first available subject as soon as a replacement was required. To meet our goal of eight cases, 14 subjects were studied. In addition to the eight valid cases, three subject were used in pilot studies, two subjects were disqualified during the training because of lack of programming skills in Java, and one subject was disqualified for not following the written instructions. Section 4.1 summarizes the characteristics of the subjects involved in the study.

3.4 Study Setting

The study was divided into four or five phases, depending on the group. To minimize potential investigator bias, each phase was described entirely through written instructions. In any phase, the subjects could ask questions, but we established guidelines restricting answers from the investigator to a clarification of the written material.

Eclipse Training Phase

To investigate the code and to perform the change, subjects were to use the Eclipse integrated programming environment for Java [9]. Because not all subjects were familiar with the Eclipse platform as a development tool, we first had the subjects complete a tutorial on how to use the principal features of Eclipse required for the study: code browsing and editing, and performing searches and cross-references. This phase was limited to 30 minutes. Subjects already familiar with Eclipse were asked to read through the tutorial, but could end the training period at their discretion. Before continuing on to the next phase, the subjects had to pass a simple proficiency test, in which the investigator asked them to perform various tasks covered in the tutorial. All subjects passed the Eclipse training test.

FEAT Training Phase

The subjects assigned to the FEAT group were required to complete a training tutorial on the FEAT tool.[3] The FEAT tool used in this study is implemented as an Eclipse plug-in: its interface is embedded in the Eclipse environment.[4] The FEAT tool allows users to create a representation for a collection of concerns, called a concern graph. Within a concern graph, a user can create individual concerns, in an iterative fashion. A concern is described by structural elements—classes, methods, and fields—that contribute to the implementation of the functionality represented by the concern. A user can place structural elements into the concern from various points in Eclipse, such as the package browser. Structural relations between elements in a concern, such as fields or methods, are detected automatically by the tool and are displayed in an Eclipse window. Within the FEAT tool, it is possible to perform queries about the structural links between different elements, in a way similar to cross-reference

[3] `http://www.cs.ubc.ca/ mrobilla/feat2`

[4] The FEAT tool described in this paper is a different implementation than the one described in a previous publication [12].

browsers. In FEAT, however, results of the queries can be accumulated in a concern representation, which serves as the context for additional queries. Elements and relations in a concern can also be viewed in a code viewer.

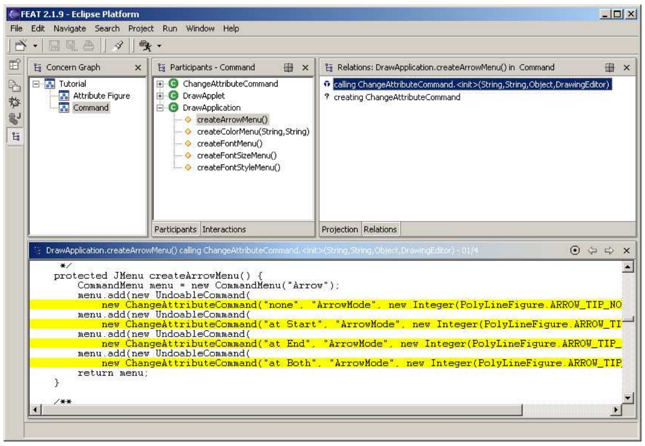

Figure 1: The FEAT Eclipse Plugin.

Figure 1 shows a concern representation in FEAT (the textual details in the figure are not important). The top left window presents the concern tree, with a root concern and two sub-concerns. The top middle window lists the program elements involved in the selected concern. The top-right window lists the relations between the selected element in the middle window and any other element in the concern. Finally, the code viewer (bottom) highlights the lines of code corresponding to the relation selected in the top right window.

The training tutorial instructed the subjects on how to use the tool effectively by focusing on one concern at a time during program investigation. The training tutorial also covered most of the features of the tool.

After completing the tutorial, subjects were asked to experiment freely with the tool. The complete training phase for the FEAT tool was limited to one hour. Before continuing on to the next phase, subjects had to pass a proficiency test, in which the investigator asked them to perform various tasks covered in the tutorial. At this stage, two prospective subjects were disqualified because they did not manage to complete the tutorial on time. These subjects were subsequently replaced.

Program Investigation Phase

After all training, the subjects were asked to read some preparatory material about the change to perform. This material included excerpts from the jEdit user manual describing file buffers and the autosave feature, instructions on how to launch jEdit and test the autosave feature, the change requirements listed in section 3.2, and a set of eight test cases covering the basic requirements. The written material for that phase also included two pointers to the code, intended to simulate expert knowledge available about the change task. These pointers consisted of the classes `Autosave` and `LoadSaveOptionPane`, the classes dealing with the autosave timer and the option pane where the autosave save frequency was set, respectively. Subjects assigned to the FEAT group were given these same pointers in the form of two pre-loaded concerns in the FEAT tool, each concern containing one class.

The subjects were then given one hour to investigate the code pertaining to the change in preparation to the actual task. The subjects were to investigate the code using the search and cross-references features of Eclipse (for the control group), or the queries of the FEAT tool (for the FEAT group). The subjects were allowed to take notes in a text file. The subjects were also allowed to execute the jEdit program, but not to change any code, even temporarily, nor to use the debugger. We set these restrictions to reduce the influence of debugging skills in Eclipse on the results. We also wanted to avoid use of print statements as a form of program understanding.

During the program investigation phase, all the activities of the subjects were recorded using a screen capture program.

Program Change Phase

In this phase, subjects were instructed to implement the requirements as efficiently as possible. They were given two hours to implement the change. Use of the debugger was again disallowed. This phase was also recorded using a screen capture program. At the end of the phase (or the two-hour period), an investigator ran through the test cases and recorded the number of test cases that succeeded. The test cases used by the investigator were exactly the same as the one provided to the subject.

Interview Phase

After the study, subjects were interviewed for 10 to 20 minutes about their experience. Questions asked by the investigator addressed the strategy they used to plan and execute their change, detailed technical questions about how some functionality was discovered and understood, and more general questions about the use of notes, and about the major problems they faced. Additionally, subjects in the FEAT group were asked how different features of the FEAT tool helped or hindered them in completing their task. The interviews were recorded using screen capture software with an audio input stream, so that pointing to various elements in the source code could be synchronized with the comments of the subjects.

4. RESULTS

We analyzed the data to assess the relative quality of the solutions produced by the subjects, and to characterize the navigation over the source code that each subject performed. To enable interpretation of this data, we also had to characterize the experience and skills of the subjects.

4.1 Subjects

Throughout the rest of this paper, subjects are identified by the following codes: C1-C4 for the subjects in the control group, and F1-F4 for the subjects in the FEAT tool group. Table 1 lists the characteristics of the subjects who took part in the study. Status indicates the status of the subject at the time of the study, either undergraduate student (Under), Graduate Student (Grad), or professional developer (Pro). The Eclipse column indicates the level of previous experience with Eclipse: subject had never used it (No), had tried

Table 1: Subject Characteristics

Subject	Status	Eclipse	Exp.	Java	Ind.
C1	Grad	Basic	6	3	3
C2	Grad	Basic	40	36	0
C3	Pro	No	62	33	34
C4	Grad	No	60	20	12
F1	Grad	Yes	41	24	28
F2	Grad	Yes	38.5	26	32
F3	Under	Yes	20	9	16
F4	Under	Yes	16.5	0.5	16

Table 2: Basic Performance Parameters

	C1	C2	C3	C4	F1	F2	F3	F4
Time (mins)	125	60	70	114	78	117	125	123
Tests (/ 8)	5	8	8	8	8	8	8	7

it but not used it on any real project (Basic), and had used previously for at least one project (Yes). The three rightmost columns in the table quantify the level of programming experience of the subjects, as reported by the subject themselves. The "Exp." column reports the total number of months of full-time programming.[5] The "Java" and "Ind." column report the number of month of Java programming experience, and programming experience in industry, respectively (both are potentially overlapping subsets of the total). We considered the experience level of these subjects sufficient for the task. As one can see from the table, random assignment of subjects resulted in more experienced subjects being assigned to the control group, but subjects with prior Eclipse experience being assigned to the FEAT group.

4.2 Change Task Solutions

Table 2 presents data about the performance of each subject. The time reported is the number of minutes taken by a subject to complete the program change phase. Time was stopped when a subject declared to be done, or after two hours (with a five-minute grace period). Most subjects used the full two hours. The tests row represents the number of the test cases passed out of a total of eight. All but two subjects, C1 and C4, passed all of the tests. These test cases represent a coarse evaluation of how well a solution met the requirements for the modification task. To characterize the solutions more precisely, we constructed a partial order that expresses an assessment of the overall quality of the solutions.

To produce the partial order, we performed an analysis of the variants in the components of the solution (*variant analysis*). This new technique to evaluate the result of a replicated software evolution task is based upon decomposing the task into different components of the solution that can be implemented differently. In our case, we divided the solution into five programming problems that the subjects had to solve to meet the change requirements. The different problems roughly correspond to the requirements of the task. For each problem we looked at the various coded solutions, and provided a partial order between each variant of the solution.

[5] The following adjustments were used for school projects: a 1-term undergrad course project counting as 0.5 month, and a 1-term graduate course project counting as 1 month.

PROBLEM 1, CHECK BOX: Adding a Check Box widget to the `LoadSaveOptionsPane` class (2 variants).

PROBLEM 2, TIMER: Stopping the timer when autosave is disabled (4 variants).

PROBLEM 3, BACKUP FILES: Deleting the autosave backup files when the autosave feature is disabled (4 variants).

PROBLEM 4, DIRTY FLAG: Resetting the autosave dirty flag for a buffer to the value of the dirty flag for that buffer (5 variants).

PROBLEM 5, RECOVERY: Preventing recovery from automatically saved backups when autosave disabled (4 variants).

For each problem, we partially ordered the solutions based on the correctness of the solutions, and on an assessment of how well the solutions respected the existing design of jEdit. As an example, consider the two variants produced for the first problem. In one variant, subjects had hard-coded the label for the check box; in the other variant, the subjects stored the check box label as a string in a resource bundle. We considered the latter solution superior to the former because it respects the existing jEdit design. Variants which we could not compare based on objective criteria were not ordered.

Table 3 compares the solutions coded by the eight subjects. Each cell in the table compares the solutions of two subjects according to the five problems identified above. The comparison for each problem is represented by a character; the characters are ordered from left to right, corresponding to problem 1 to 5, respectively. A $-$ indicates that the variant implemented by the subject identified by the row is worse than the one implemented by the subject identified by the column. A $+$ indicates that the variant implemented by the subject identified by the row is better than the one implemented by the subject identified by the column. An equal sign ($=$) indicates that the two subjects have used the same variant, and a question mark (?) indicates that the two subjects have used variants that cannot be compared objectively.

Considering each problem as having equal weight, we can use the characterizations given in table 3 to derive a partial order on the complete solutions. Removing equal signs and question marks, and cancelling out $-$ and $+$ signs leaves a number of positive or negative signs for each pair of subjects. The resulting partial order is show in figure 2, which shows C1's solution as the worst, and C3's as the best.

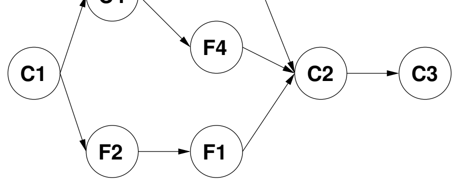

Figure 2: Partial Order on the Quality of Solutions.

Table 3: Partial order on solutions

Subjects	C2	C3	C4	F1	F2	F3	F4
C1	= − − −−	= − − −−	+ − − − −	+ − − − −	+ − − − −	+ − − − −	= − − −?
C2		=== − =	+ + ++?	+ ===?	+ + ++?	+ ===?	==== +
C3			+ + ++?	+ == +?	+ + ++?	+ == +?	=== ++
C4				= − − − =	=??? =	= − − − =	− − − − +
F1					= + + + =	=====	− === +
F2						=====	− === +
F3							− === +

4.3 Code Navigation

For each case, we recorded the activities of the subject during both the program investigation and program change phases. To determine what parts of the program a subject investigated, and how the subject found those parts, we have transcribed the screen capture recordings into a list of *transition events*, each of which representing a switch from one program element (typically, a field or method declaration) to another. The information recorded for each transition event consists of the time-stamp of the event, the two program elements involved in the transition, and the mechanism used to perform the transition, such as selecting an element in the code browser, performing a search, and accessing the API documentation, among others. To qualify as a transition event, the switch between elements had to involve the examination of the second element (the element transitioned to). We defined "examination" as the element appearing in the recording long enough for a subject to be able to at least read it once. For example, accessing the declaration of a method through the code browser and having the declaration displayed on the screen for 20 seconds counted as a transition to the method. Flashing the result of a query on the screen for one second during a systematic traversal of a list of query results did not. Transitions between members in the two classes given to the subjects as part of the study set-up (`LoadSaveOptionPane` and `Autosave`) were not recorded because the two classes are small enough that their entire contents can be displayed on the screen, and because these classes were examined unusually often, possibly because of their special status in the study. Each transition event has been categorized as either *structural* (the transition is the result of a structural relation in the code), *local* (the transition is the result of locality in source files), *browsed* (the transition was performed by selecting an element from the code browser, or *recalled* (an element previously considered was accessed again by returning to the previous view). In practice, these categories imply slightly different, but roughly equivalent, user-interface actions, depending on whether they were performed in the Java Perspective of Eclipse, or through the FEAT tool.

Based on transcripts of the transition events, we have produced a *navigation graph* for each subject. We use the graphs to determine how much of the total code examined for the task was discovered in the investigation phase, to help us qualify how systematic the behavior of subjects during the program change was, and as a basis to infer potential concerns considered during the investigation phase. To provide a basis for interpreting the metrics we analyze form these graphs, figure 3 illustrates the navigation of subject C2 during the subject's investigation phase. Each node represents an element examined during the phase, and numbered edges represent the sequence in which each node was examined. We first report on the quantitative characterization of each graph, and then discuss how we inspected each graph to infer concerns during the investigation phase.

Table 4: Metrics of Program Investigation

Subject	N	E*	L	B	S	R
C1	8 (32)	11	5%	26%	11%	58%
C2	28 (37)	27	2%	27%	44%	27%
C3	27 (35)	32	2%	27%	46%	25%
C4	29 (40)	26	9%	12%	45%	34%
F1	26 (30)	17	5%	26%	56%	14%
F2	17 (27)	18	3%	26%	26%	46%
F3	23 (35)	25	0%	17%	67%	17%
F4	28 (51)	50	3%	10%	55%	32%
Mean	23 (36)	26	4%	21%	44%	32%

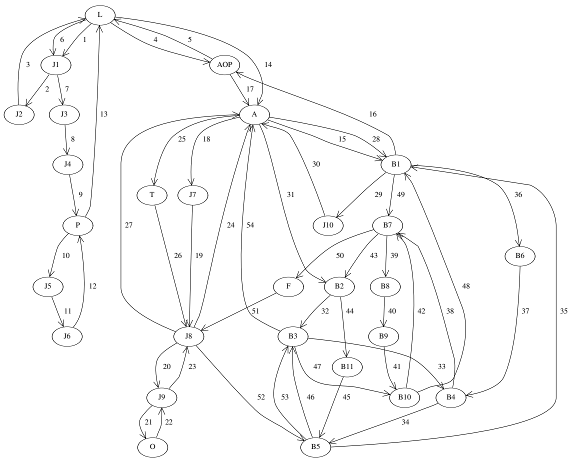

Figure 3: Navigation graph for subject C2 (investigation phase)

Quantitative Analysis

Table 4 reports metrics on the navigation graph of each subject during the program investigation phase. For each subject, the table first presents the number of nodes in the navigation graph (N) with the total number of nodes examined for both phases in parentheses. These numbers provide insight into the relative proportion of code explored in the investigation phase. The next column lists the number of repeated edges (E*). This number represents the number of transitions above the number of transitions required to visit every element once. As part of the research, we assumed that a high number of repeated edges is indicative of an unclear understanding of the code. The last four columns show the proportion of edges representing local transitions (L),

Table 5: Metrics of Program Change

Subject	N	E*	L	B	S	R
C1	26 (32)	48	19%	23%	9%	49%
C2	16 (37)	16	0%	28%	47%	25%
C3	17 (35)	33	10%	20%	28%	42%
C4	22 (40)	61	4%	10%	30%	57%
F1	16 (30)	12	4%	50%	36%	11%
F2	15 (27)	40	7%	18%	13%	62%
F3	23 (35)	40	2%	21%	32%	46%
F4	32 (51)	98	3%	27%	23%	47%
Mean	21 (36)	44	6%	25%	27%	42%

browser transitions (B), structural transitions (S), and recall transitions (R).[6] Table 5 summarizes the same data for the program modification phase.

Qualitative Analysis
To evaluate our study hypotheses, we needed to assess the concerns that a subject systematically considered when performing the task. We inferred the concerns by triangulating data from the navigation graphs from the investigation phase, the notes and concern graphs produced, and the interview data. Originally, we had planned to use the interview as a primary source of data to glean insight into the concerns a subject had considered. However, a detailed analysis of the interview transcripts revealed that subjects had poor recall of their behavior during the task: statements made by a subject often did not match the transcripts of their actions. To avoid this suspect data, we used the responses in the interviews only as support for concerns we inferred through other data. We based our concern inferences only on data from the investigation phase. Inferring concerns from the change phase was problematic because the phase involved coding, and because some subjects worked less systematically as they had less time available to complete the task. Space constraints prevent us from elaborating on how we inferred every concern. Our overall concern inference strategy was to look in navigation graphs for continuous sequences of semantically and structurally related elements. In these cases, if we also found evidence of the elements examined in a navigation sequence in the notes, concern graph, or interview, we considered the sequence to be a concern. As an example, in the recording of subject C2's investigation phase, we see the subject writing in the notes the question, "How do we delete all of the autosave files?", followed by a series of navigation events, and then the subject appending information to the notes file to summarize his discoveries. In our analysis, this constituted a concern.

Table 6 lists, for each subject, the concerns inferred in the investigation phase, including the number of program elements examined in the context of the concern, and the sources of evidence for the inference (G for the navigation graph and screen recording, N for notes file, C for concern graph, and I for interview).

The general investigation strategies used by subjects can be summarized as follows. C2 and C3 stepped systematically through each requirement, investigating the code relevant to each requirement separately, even when code for different concerns overlapped. C1 read the details of the code for the two classes given, C4 investigated the methods required for a posited change, and all the subjects in the FEAT group explored the code of the system with no explicit goal (as could be determined by our data), noting potentially pertinent sections of code as they were traversed.

[6]The percentages do not always add up to 100 because of rounding.

Table 6: Concerns inferred (investigation phase)

Subject	Concern	Size	Evidence
C1	Properties handling	4	G
C2	Properties handling	7	G,N
	Autosave interval change event	5	G,N,I
	Autosave file deletion	14	G,N
	Autosave recovery code	2	G,N,I
C3	Autosave file deletion	10	G,N,I
	Autosave interval change event	4	G,N,I
	Autosave recovery code	4	G,N
C4	Properties handling	6	G,N
F1	None		
F2	Properties handling	5	G,C,I
F3	Autosave recovery code	6	G,C
F4	Autosave interval change event	3	G,C,N

5. ANALYSIS

Because of our small number of cases, our evaluation of the research hypotheses relies on a logic of replication influenced by case study research methods. Using this logic, we describe how the data collected during the study supports the hypotheses. Then, we discuss the threats to the validity of our study.

5.1 Replication Logic

In this research, we use the method of "analytic generalization" to interpret our results, regarding each change task executed by a subject as a "case" in a multiple-case study. With analytic generalization,

> ...a previously developed theory is used as a template with which to compare the empirical results of the case study. If two or more cases are shown to support the same theory, replication may be claimed. [20, p. 31].

Our analysis aims to demonstrate that there is evidence supporting the two hypotheses H1 and H2 proposed in section 2.

5.2 Evaluation of the Hypotheses

In evaluating the support for the proposed hypotheses, we have chosen to use data from the most successful and least successful subjects. Evidence of a successful subject using a concern-oriented strategy involving structural relations would support our hypotheses, while evidence of an unsuccessful subject using a concern-oriented strategy involving structural relations would contradict our hypotheses. Moderately successful subjects will be left out of the analysis, since it is difficult to interpret their contribution to the evaluation.

We used three independent sources of data to assess a subject as successful or not. First, we considered the placement of the coded solution of the subject in the partial order of all the solutions (Figure 2). Operationally, this meant that we assigned a value for each subject that corresponded to the number of solutions below that solution in the partial order. This analysis yields the following sequence: 0 1 1 2 2 2 6 7. We considered solutions with order 0 and 1 as unsuccessful, and 6 and 7 as successful. Second, we considered the time taken to complete the task (Table 2). Time of completion can be indicative of a successful change because the subjects were instructed to perform the change as efficiently as possible. This data shows that the two best solutions also correspond to the two subjects who took the least time to complete the task. Finally, we took into account an assessment of how systematic an approach was taken in the change phase. Operationally, this was determined through the number of repeated edges in the navigation graph for the change phase (Table 5). The subjects for the cases with order 6 and 7 also exhibited an approach that was more systematic than average. Based on these sources of evidence, we will consider C2 and C3 the successful subjects. In contrast, the three worst coded solutions (by subjects C1, C4, and F2) also correspond to times above 114 minutes (practically the entire allocated time), and the second, third, and fourth less systematic changes, respectively. To determine a subject's uses of structural queries, we inspected the relative importance of structural queries in the subject's navigation,[7] considered whether there was evidence of structural relations in the notes, concern graphs, or interview responses.

Subject C2 (Successful)
During the investigation phase subject C2 examined 76% of the total program elements examined during the task. During the investigation phase the subject focused separately on four scattered concerns. These concerns were determined from the screen recording: the subject had typed notes describing questions of interest before investigation theses questions in the code. In the interview, C2 described proceeding systematically through the modification task requirements, investigating sections in the code that corresponded to specific questions (concerns) for each requirement. The subject's program change phase was the second most systematic, with only 16 repeated edges. All the above evidence supports hypothesis H1. C2 also used more structural queries than any other kind (60%), and stored some of the information discovered as call chains in the notes taken. This evidence supports H2.

Subject C3 (Successful)
Subject C3 covered 77% of the total elements considered during the investigation phase. During this phase, the subject also focused on three overlapping concerns separately. These were based on evidence from the screen recording, such as reading a specific requirement, investigating scattered code addressing this requirement, and subsequently writing notes describing the implementation of that concern. The resulting program change phase was the third most systematic, with 33 repeated edges, and C3's solution was the best. This evidence supports H1. C3 also used more structural queries than any other kind (61%); evidence that supports H2.

Subject C1 (Unsuccessful)
During the investigation phase, subject C1 covered very little of the source code, focusing instead on the two provided classes. Evidence of only one concern was present, the investigation of this concern was limited (4 nodes), and the subject did not take any notes to document the concern. The program change phase was also less systematic than average, with 48 repeated edges. C1 used the least amount of structural queries of all subjects (25%). To contradict H1, an unsuccessful subject should have demonstrated a concern-oriented strategy. There is no evidence of such a strategy in the case of C1, so H1 is not contradicted. The same applied to H2.

Subject F2 (Unsuccessful)
During F2's investigation phase, we found evidence of a focus on only a single concern, Properties handling. This concern is the least scattered in the code, and is the closest structurally to the classes provided as clues. In the interview, the subject indicated that the general strategy used was "just trying to find out [...] what's going on...", without any specific question in mind. Subject F2 also used relatively few structural queries (47%). The evaluation in this case is similar to C1: H1 is not contradicted.

Subject C4 (Unsuccessful)
During the investigation C4 examined a high proportion of the total elements considered during the task (73%), but did not seem to follow any concern-oriented strategy. The only evidence of concern focus was for the properties handling concern. The corresponding change task was very unsystematic (with 61 repeated edges), and the subject mentioned in the interview about "being lost" and "taking shots in the dark". The evaluation in this case is similar to C1: H1 is not contradicted.

Conclusion and future work
We conclude that there is evidence that concern-oriented program evolution was associated to success in our study. However, the results also show that, contrary to our expectations, explicit tool support for concern-oriented program investigation did not provide an advantage. Although the subjects in the FEAT group were explicitly instructed to focus on a single concern at the time, they did not do so. We hypothesize that external factors, such as skill and experience, may have been more important in influencing concern-oriented code investigation than tool support. The recorded sessions and interviews show evidence that the subjects in the FEAT group did not focus on concerns because it was not immediately clear to them what constituted a concern. These observations seem to agree with earlier results by Wiedenbeck *et al.* indicating that "expert programmers make explicit mappings between segments of program code and the subgoals that they implement" [19, p.805]. In our research scattered concerns can be associated with subgoals. There is also evidence that structural queries played an important role in helping subjects carry out their task.

We make three observations based on the results of this study.

[7] The proportions we calculated exclude recall category, because recalled transitions do not reflect the acquisition of new knowledge about the program. The percentages used here can be obtained from table 4 using the formula:
$S/(L + B + S)$.

1. A concern-oriented strategy for program evolution was more successful than other strategies.

2. Tool support for a concern-oriented program evolution strategy does not compensate for experience.

3. Although use of structural queries is associated with successful cases, such querying is not effective in the absence of a strategy for performing an evolution task.

These results increase our confidence in the value of concern-oriented program evolution. These results also indicate that tool support for concern-oriented program evolutions needs to be coupled with adequate background and training to be effective. Finally, the results suggest future avenues of research. First, we plan to examine in more detail the transcripts and other data we collected to determine how to best help developers focus on the important code relevant to concerns, and to record this knowledge so that it can be used effectively during coding. Second, we are currently working on algorithms to support the automatic determination of concerns of interest based on navigation graphs. Finally, we plan to replicate this study with improved tool support and subjects more familiar with the concept of scattered concerns.

5.3 Experimental Critique

Construct Validity
In establishing operational measures the most important analyses for our study were determining the level of success of each subject, and determining the concerns considered during the investigation phase. To help ensure valid results we have used multiple sources of evidence for these measures [3]. When evaluating the level of success of each subject we used three independent sources of evidence: the coded solution, the time taken by subjects, and the number of repeated edges in the navigation graphs that characterize the subject's change phase actions. When inferring the concerns on which subjects focused during investigation, we used the subject's navigation graph and additional clues from the screen recording data, the interviews, and the notes and/or concern graph.

Internal Validity
The internal validity of our study is threatened by the possibility that the success level for a case is determined by a different, competing factor, such as prior knowledge, proficiency with the development environment, and investigator bias during the study. To reduce this possibility we took steps to ensure that no subject had prior knowledge of jEdit, we asked subjects not to communicate the details of the study to others, we provided basic training with Eclipse to each subject, we precluded the use of powerful features of Eclipse, such as the debugger, and we scripted the entire study, limiting the role of the investigator to answering questions. There is always the possibility of investigator bias in the answers to the subject's questions. To limit this effect we established guidelines at the start of the study for the investigator to use in answering questions: the investigator was to only answer questions about the features of the tools covered in the tutorial, and provide no comment about the task. There are many potential sources of bias related to the interview data. With interviews, bias is possible both from the subjects (wanting to provide the "right" answer), and from the investigator (wanting to elicit or find the "right" answer). For this reason, we have treated the interviews as an unreliable source of evidence, using such data only to support other sources of evidences, and only when all other sources agreed.

External Validity
The applicability of our study's findings must be carefully established. All of the subjects were either students or recent graduates from a computer science department. Different results might be obtained from a population of senior developers not formally trained in object-oriented programming. Another threat to the generality of our study is our use of a single task. Although the scope of our study is explicitly limited to evolution tasks involving some form of scattered concerns, there exist many different evolution problems. We do not expect that similar results can be obtained from all problems. In particular, evolution tasks involving very complex algorithms might yield different results. Nevertheless, we believe that our use of a real task in a program large enough that it can not be understood in a short amount of time, contributes to achieving an acceptable level of external validity.

Reliability
All the operations of in this study, including the data collection procedures, have been documented and will be made public on our web site. The task was defined on an open-source code base. As a result, it should be possible to replicate the study.

6. RELATED WORK

Many empirical studies of programmers have been reported in the literature. We discuss here the previous work closest to ours.

Letovsky *et al.* [8] reported that professional programmers have difficulty understanding programming concepts scattered in different parts of the program ("delocalized plans"), and suggested a documentation strategy to address the problem. In contrast, our work assumes the presence of delocalized plans, which we call scattered concerns, and the focus of our investigation is to evaluate an effective strategy to address such concerns.

Wiedenbeck, Corritore, and others carried out many studies to investigate various characteristics of the mental representations of programmers during software maintenance tasks. These include studies of the differences between expert and novices [19], and between procedural and object-oriented programmers [4, 5]. However, these studies targeted very small programs (135 to 822 loc). As a result, the subjects involved in the studies were able to manage and understand a significant fraction of the code. This situation might lead to behavior that differs significantly from behavior exhibited when it is only possible to investigate a very small faction of a large code base, as in our study.

Using the technique of protocol analysis, von Mayhauser, Vans and others studied the comprehension processes of programmers during large-scale corrective and perfective maintenance [17, 16, 15]. The focus of the work was to investigate the cognitive processes of programmers. As such, the studies do not include a detailed analysis of the result of the maintenance activities. The aim of our work is to investigate whether a specific type of strategy, concern-oriented program investigation, can help developers during program evolution involving scattered concerns.

Baniassad *et al.* conducted a study of eight industrial and academic developers involved in a software evolution task, to investigate the kinds of scattered and overlapping ("crosscutting") code that developers encounter, and how this code is managed. Based on a series of interviews with each participant, the researchers observed that crosscutting concerns usually emerge as obstacles to be overcome, and that the strategy used to cope with the obstacles tends to vary based on the nature of the obstacle. The study does not involve an evaluation of the results of the program evolution activities so it is not possible to determine precisely how effective different strategies are in managing scattered concerns.

Finally, Walker *et al.* performed two exploratory experiments to study the impact of aspect-oriented programming on common programming tasks [18]. The experiments involved a small number of programmers working on two maintenance tasks, one corrective and one perfective. The results provide key insights about the ease of code understanding in the presence of aspect-oriented concepts. The analysis of the program evolution sessions also show evidence of a concern-oriented strategy of program evolution.

7. SUMMARY

Program evolution tasks often involve addressing different conceptually-related segments of the implementation (concerns), which can be scattered across multiple modules. Existing tools proposed to help address scattered concerns assume that developers can follow a concern-oriented program investigation strategy when evolving software, and assume that this strategy is effective. To investigate these assumptions, we have carried out a study of a software evolution task performed both with and without tool support for concern-oriented program investigation. The study provided evidence to support the hypothesis that a systematic investigation of code that addresses a single concern at a time leads to more successful completion of an evolution task. The study also showed evidence that navigating code along structural relations between program elements can help to capture scattered knowledge about a concern. Our analyses of the data from the study yielded two novel analysis methods: navigation graphs, which support the analysis of a subject's behavior when investigating source code, and variant analysis, which we used for evaluating the results of a program evolution task. In addition to the evaluation of our hypotheses, the analysis of the study data showed that tool support for a concern-oriented program evolution strategy does not compensate for experience, and that the use of structural queries when investigating a program is not effective in the absence of a strategy for performing an evolution task. Whether the FEAT tool we have used in this study is an effective way to support concern-oriented program investigation is still an open question, which we could not answer based on our study data. In the future, we plan to further our understanding of the factors contributing to success at program evolution tasks through additional studies addressing more specific aspects of tool support.

ACKNOWLEDGMENTS

The authors would like to thank Davor Cubranic, Chris Dutchyn, Joanna McGrenere, and Adam Murray for their thorough and insightful feedback on early versions of this paper, and Dima Brodsky for his help with the management of the study data. The study was funded by an NSERC research grant, a University of British Columbia Graduate Fellowship, and the IBM Corporation.

REFERENCES

[1] E. L. Baniassad and G. C. Murphy. Conceptual module querying for software reengineering. In *Proceedings of the 20th International Conference on Software Engineering*, pages 64–73, May 1998.

[2] E. L. Baniassad, G. C. Murphy, C. Schwanninger, and M. Kircher. Managing crosscutting concerns during software evolution tasks: An inquisitive study. Proceedings of the 1st Conference on Aspect-Oriented Software Development, April 2002.

[3] L. Bratthall and M. Jørgensen. Can you trust a single data source exploratory software engineering case study? *Empirical Software Engineering*, 7(1):9–26, March 2002.

[4] C. L. Corritore and S. Wiedenbeck. Mental representation of expert procedural and object-oriented programmers in a software maintenance task. *International Journal of Human-Computer Studies*, 50(1):61–83, January 1999.

[5] C. L. Corritore and S. Wiedenbeck. An exploratory study of program comprehension strategies of procedural and object-oriented programmers. *International Journal of Human-Computer Studies*, 54(1):1–23, January 2001.

[6] W. G. Griswold, J. J. Yuan, and Y. Kato. Exploiting the map metaphor in a tool for software evolution. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 265–274. ACM, May 2001.

[7] D. Janzen and K. De Volder. Navigating and querying code without getting lost. In *Proceedings of the Conference on Aspect-Oriented Software Development*, March 2003. To appear.

[8] S. Letovsky and E. Soloway. Delocalized plans and program comprehension. *IEEE Software*, 3(3):41–49, May 1986.

[9] Object Technology International, Inc. Eclipse platform technical overview. White Paper, July 2001.

[10] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[11] D. L. Parnas. Sofware aging. In *Proceedings of the 16th International Conference on Software Engineering*, pages 279–287, 1994 1994.

[12] M. P. Robillard and G. C. Murphy. Concern Graphs: Finding and describing concerns using structural program dependencies. In *Proceedings of the 24th International Conference on Software Engineering*, May 2002.

[13] M. Sanella. *The Interlisp-D Reference Manual*. Palo Alto, USA, 1983.

[14] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton. N degress of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering*, pages 107–119. ACM, May 1999.

[15] A. M. Vans, A. von Maythauser, and G. Somlo. Program understanding behavior during corrective maintenance of large-scale software. *International Journal of Human-Computer Studies*, 51(1):31–70, July 1999.

[16] A. von Mayhauser, A. M. Vans, and A. E. Howe. Program understanding behaviour during enhancement of large-scale software. *Journal of Software Maintenance: Research and Practice*, 9(5):299–327, September/October 1997.

[17] A. von Mayrhauser and A. Vans. Identification of dynamic comprehension processes during large scale maintenance. *IEEE Transactions on Software Engineering*, 22(6):424–437, 1996.

[18] R. J. Walker, E. L. Baniassad, and G. C. Murphy. An initial assessment of aspect-oriented programming. In *Proceedings of the 21st International Conference on Software Engineering*, pages 120–130, May 1999.

[19] S. Wiedenbeck, V. Fix, and J. Scholtz. Characteristics of the mental representations of novice and expert programmers: an empirical study. *International Journal of Man-Machine Studies*, 39:793–812, 1993.

[20] R. K. Yin. *Case Study Research: Design and Methods*, volume 5 of *Applied Social Research Methods Series*. Sage Publications Ltd., London, second edition, 1989.