In *Proceedings of the 18th International Conference on Automated Software Engineering, pp. 225–234.*
IEEE Computer Society Press, October 2003. ©IEEE, 2003

Automatically Inferring Concern Code from Program Investigation Activities

Martin P. Robillard and Gail C. Murphy
Department of Computer Science
University of British Columbia
2366 Main Mall, Vancouver, BC
Canada V6T 1Z4

Abstract

When performing a program evolution task, developers typically spend a significant amount of effort investigating and re-investigating source code. To reduce this effort, we propose a technique to automatically infer the essence of program investigation activities as a set of concern descriptions. The concern descriptions produced by our technique list methods and fields of importance in the context of the investigation of an object-oriented system. A developer can rely on this information to perform the change task at hand, or at a later stage for a change that involves the same concerns. The technique involves applying an algorithm to a transcript of a program investigation session. The transcript lists which pieces of source code were accessed by a developer when investigating a program and how the different pieces of code were accessed. We applied the technique to data obtained from program investigation activities for five subjects involved in two different program evolution tasks. The results show that relevant concerns can be identified with a manageable level of noise.

1. Introduction

When performing a program evolution task, developers expend a significant amount of effort investigating source code. The goal of this investigation activity is typically to answer specific questions about the implementation of various mechanisms affecting the modification. For example, before undertaking the addition of a feature to a graphical user interface, a developer may want to identify where menus are created, and how commands are passed to the different menu items. As another example, in a database system, a developer may need to understand an undocumented protocol for accessing a record object. Finding the answers to such questions, or concerns, requires examining source code that is typically scattered in many locations. In a large system, developers can realistically examine only a small subset of the source code as part of a program evolution task. Determining which pieces of the code to examine is a complex problem that cannot be solved automatically with source code analysis techniques. Typically, developers determine which source code to examine through a combination of reasoning and informed guesses that build on their skills and experience. This process is naturally fraught with errors and dead-ends. Successive iterations are typically needed to identify the pieces of source code relevant to the task.

There are several advantages to documenting hard-earned knowledge about the implementation details of various concerns. For instance, the developer performing the change can use the documentation to help make the modification systematically and in a robust fashion [4, 7, 10]. In the longer term, other developers performing similar changes may be able to use the knowledge without needing to perform all of the time-consuming program investigation. However, concerns are rarely documented because it is difficult for a developer to know how to document the concerns investigated, and the potential benefits may not outweight the costs of documenting the concerns.

To address this problem, we present a technique to automatically infer the essence of the program investigation performed by developers, so that the results of this activity can be documented as a set of concern descriptions at a negligible cost. The concern inferencing algorithm we describe extracts a user-specified number of elements from all of the elements considered during the program investigation. It then groups these elements into clusters representing potential concerns. To document concerns stemming from a program investigation task, a developer presented with the results of our technique has only to invalidate useless clusters, and to name and save useful ones.

The algorithm is based on what source code a developer examined during a program investigation session, and on how the developer moved between different pieces of source code. The algorithm takes as input a program investigation transcript obtained by recording, in sequence, every change in the source code that is visible to a developer, and the cause for the change (e.g., selecting an element in a code browser, viewing the result of a search, etc.). The inference algorithm takes into account a variety of factors, namely, the order of program elements in the sequence, the way in which elements were accessed, and whether there exists in the source code structural program relationships between the elements examined. For a specified number of program elements, the algorithm produces a set of clusters that constitute candidate concerns.

We applied our algorithm to data obtained from two different evolution tasks. Each task was replicated with different developers. We found, not surprisingly, that results varied between developers and tasks. However, in all cases, we were able to obtain concerns describing interactions relevant to the change task out of hundreds of elements examined during the investigation.

The rest of this paper is structured as follows. In Section 2, we present an example motivating our research. In Section 3, we describe the format of the investigation transcripts we use as the input to our algorithm. In Section 4 we describe our inference algorithm. In Section 5, we report on the concerns obtained by running our algorithm on data obtained from two evolution tasks, and discuss the influence of various factors on the results. We discuss related work in Section 6 and conclude in Section 7.

2. Motivation

To illustrate the motivation and benefits of our research, we use the example of a program evolution task on a text editor application. In this application, any changed and unsaved 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 through the execution of a menu command in the application's menu bar. If the application crashes with unsaved buffers, the next time it is launched, it will attempt to recover the unsaved files from the automatically-saved backups. A user can disable the auto-save feature by specifying the auto-save frequency as zero. However, this option is undocumented, and can only be discovered by inspecting the source code.

One day, Bob is asked to modify the application to support the disabling of the auto-save feature explicitly through a check box in the graphical user interface. Since the code base for the text editor is much too large to analyze as a whole, Bob focuses on understanding how the auto-save feature is implemented. In doing so, he asks himself questions, such as "how is the auto-save command timed?". He then proceeds to investigate the code to answer these questions (or concerns). This investigation might involve false turns and the examination of unrelated code, but in the end, it will (hopefully) involve examining source code that answers his questions. In our case, let us assume that the implementation of the timing of the auto-save command involves five methods scattered in three different classes. Because it is a fairly simple mechanism, Bob makes a mental note of it, and proceeds to investigate another question related to the change. After completing the modification, Bob moves on to another project and forgets much of what he has learned about the auto-save feature.

Months later, Alice is asked to modify the auto-save feature to use a new library that supports the auto-saving of objects. At some point in the preparation for the change, she wonders: "How is the auto-save command timed?" From the bug database for the project, Alice determines that Bob had previously performed a change related to the auto-save feature. Alice thus asks Bob for a short briefing. Unfortunately, the only thing Bob remembers about the change is that there is some class called `Autosave` that is involved in it. To understand the actual details of how the auto-save feature is implemented, Alice has to spend time investigating the code, in effect re-doing some of Bob's work.

A more desirable situation would be for Alice to be able to start with some description of how the auto-save command is timed in the text editor. With our concern inference algorithm, we envision the following scenario. While Bob investigates the program, the integrated development environment he uses logs the code he examines, and the various commands he uses to navigate between different areas of the code. When he is done and commits the change, a concern inference algorithm automatically executes on the transcript of his program investigation and opens a window listing, for example, three sets of methods representing potential concerns he investigated. Bob does not see the relevance of two of them but the third one lists five methods involved in the implementation of the auto-save command. Even though Bob only ended up modifying one of the five methods, he decides that the five methods implement the concern. He names the concern "Code to time the auto-save command", and saves it in a database of concerns. The whole procedure takes less than one minute. Developers working on subsequent tasks that involve the auto-save feature can now query the database to determine what Bob considered relevant to the timing of the auto-save command. Furthermore, this information is more descriptive than the single method affected by the change, which would be available by searching through the revision control system for the project.

3. Investigation Transcripts

The inference of concerns from program investigation activities requires a transcript of the operations performed by a developer. Informally, a transcript records all of the source code visible to a developer during a program investigation session, and the sequence in which different areas of the code are viewed. In discussing the areas of source code under consideration by a developer, our unit of granularity is the method declaration and, in some cases, field declaration. Other elements normally present in source code, such as class declarations and comments, are not considered. We chose this approach because the concerns inferred by our algorithm are expressed only in terms of class members.

For our purpose, we formally define a *program investigation transcript* as an ordered set of *investigation events* $E = \{e_1, ..., e_n\}$. An event corresponds to a change in the set of method declarations visible to a developer. We define a method declaration as visible if it is completely or partially visible in the *active* editor window of a software development environment. If multiple editor windows are visible, then only the one with the focus of the windowing system is considered visible. Because, in many cases, all field declarations can appear at once to a developer, we did not consider it useful to include field declarations as a part of the transcript, except in special circumstances described below.

An event e consists of a tuple (D, c, X). The set D lists identifiers for all of the method declarations (and certain field declarations) visible immediately after the event. The element c is a category value describing what caused the event. It can take the following values:

- **B**: the content of the active editor changed as the result of selecting an element in a code browser.
- **C**: the content of the active editor changed as the result of following a cross-reference between two elements.
- **R**: an editor window was recalled from an existing buffer of visible windows, such as a history list or tabbed pane.
- **L**: the content of the active window changed as the result of scrolling up and down in a file.
- **K**: the content of the active window changed as the result of a keyword search.

The last tuple element, X, is an ordered set of elements representing extra information about the event. For browser events (**B**), X contains a single identifier representing the declaration that was accessed through the browser. For example, if a developer selects method M2 from a browser window and reveals co-located methods M1 and M3 in addition to M2, then the event would be transcribed as $(\{M1, M2, M3\}, B, \{M2\})$. For a cross-reference event (**C**), $X = \{x_1, x_2\}$ contains the identifiers of both the domain (x_1) and the range (x_2) of the cross-reference. For a keyword event (**K**), X contains an identifier representing the declaration in which the keyword was found. For all other events, $X = \emptyset$. For browser, cross-reference, and keyword events, if the set X contains a field declaration, then this declaration is included in the set D. Otherwise, fields declarations are ignored.

During the investigation of a program, a new event e is created every time the set D of visible elements changes. Figure 1 shows an example of a segment of investigation transcript. The first line shows an event corresponding to method B137 being revealed as the result of a keyword search. The next event corresponds to methods F29, F30, and F31 being revealed as a result of accessing method F30 through a cross-reference from B137 (F29 and F31 are also visible because they are co-located with F30). Method B137 is then recalled from a previous view. Then field B24 is displayed through a browser access (with co-located method B167). Finally, the file is scrolled to reveal B168 and hide B24.

```
B137            K   B137
F29,F30,F31     C   B137,F30
B137            R
B24,B167        B   B24
B167,B168       L
```

Figure 1. Example investigation transcript

4. Inference Algorithm

Given a program investigation transcript, our aim is to automatically extract potential *concern descriptions*. At this stage, what we signify by the term concern description is a group of program elements (method and field declarations) that are related in the investigation, and that are potentially involved in the implementation of an actual user-level concern. For simplicity, when it is clear from the context whether we are referring to a user-level concern or a concern description, we shall use the term concern interchangeably. We propose an algorithm that can generate concern (descriptions) based on a calculation of how related different elements were during a program investigation session. Our concern inference algorithm is divided in three phases. A first phase assigns, to each element in the set D of every event, a probability that this element was actually examined by the developer. A second phase calculates a metric of correlation for every pair of elements in the transcript. The third phase generates a set of concerns based on the correlation metric calculated in the second phase.

4.1. Calculating Probabilities

As we mentioned in Section 3, to each event e_i corresponds a set D_i of method (or field) declarations visible to the developer. However, at any point of the investigation,

```
1:  for all $e_i = \{D_i, c_i, X_i\} \in E$ do
2:      for all $d_{i,j} \in D_i$ do
3:          $w_{i,j} \leftarrow 1$
4:          if $(c_i = \mathbf{B} \vee c_i = \mathbf{K}) \wedge d_{i,j} = x_{i,1}$ then
5:              $w_{i,j} \leftarrow w_{i,j} + \alpha$
6:          else if $c_i = \mathbf{C} \wedge d_{i,j} = x_{i,2}$ then
7:              $w_{i,j} \leftarrow w_{i,j} + \alpha$
8:          end if
9:          if $c_{i+1} = \mathbf{C} \wedge d_{i,j} = x_{i+1,1}$ then
10:             $w_{i,j} \leftarrow w_{i,j} + \alpha$
11:         end if
12:     end for
13: end for
```

Figure 2. Calculating probabilities

the developer was not necessarily examining each one of the declarations in the corresponding set D_i. To account for the fact that, at each event, the developer is probably focusing on only one or two of the visible declarations, we assign a probability to every element $d_{i,j}$ of the set D_i of every event.

We determine the probability of an element being examined by first assigning a weight $w_{i,j}$ to each element. The weight for an element is based on the category of event (c) and the additional information X. Certain conditions, as expressed in the algorithm of Figure 2, increase the weight of an element by a confidence parameter α.

Informally, the weight of an element in an event is increased if the element is the same as the element in the extra information set (X) in a browser or keyword event (lines 4–5), or if the element is the same as the range element of a cross-reference event (lines 6–7). Additionally, the weight of an element is increased if it is the domain of a following cross-reference event (lines 9–10).

Once all the weights are calculated, we can determine corresponding probabilities.

$$p(d_{i,j}) = \frac{w_{i,j}}{\sum_{k=1}^{n} w_{i,k}}$$

For example, using $\alpha = 5$, the probabilities for the second event in figure 1 are: $p(\text{F29}) = \frac{1}{8} = 0.125$, $p(\text{F30}) = \frac{6}{8} = 0.75$, $p(\text{F31}) = \frac{1}{8} = 0.125$.

4.2. Calculating the Correlation Metric

Our algorithm infers concerns by analyzing the correlation between different pairs of elements potentially examined by a developer. The intuition behind this idea is that if a developer focuses on a pair of elements, then there is a possibility that the relations between the two elements in the pair bears an important significance to the task. Thus, the underlying principle of our concern inference algorithm is to determine how strongly different pairs of elements are related in the context of the program investigation. To do so, the algorithm takes the set of all elements present in the transcript, analyses every possible combination of two elements, and assigns a correlation metric to each pair. The correlation metric is based on an analysis of how close two elements are in the investigation sequence, the category of event for each element, and whether the elements are directly related in the program (for example, through a method call). The analysis also takes into account the probabilities calculated for each element.

The correlation algorithm is configured through nine parameters, β_0, β_1, β_2, β_B, β_C, β_R, β_L, β_K, β_S, and one function on the program investigated, $related(x, y)$. The first three parameters weight the importance of two elements being displayed consecutively (β_0), or being separated by only one (β_1), or two (β_2) elements. The next five parameters are factors weighting the importance of different event categories on the investigation. For example, an element revealed as the result of scrolling (**L**) might not be as significant as an element revealed through a cross-reference (**C**). The parameterization allows flexibility in determining this importance. The last parameter, β_S, factors in the importance that two elements be *actually* related in the program. This is determined by the function $related(x, y)$, which returns true if there is a direct structural link between x and y in the program. For two elements (field or method) x and y, *related* returns true if

- x calls y (or vice-versa),
- x accesses (field) y, or
- x implements or overrides y (or vice-versa).

The algorithm we use to generate the correlation metric $m_{i,j}$ between two elements is presented in Figure 3. This algorithm first determines the list of all elements revealed during the program investigation (line 1). For every unordered pair of elements (lines 2–3), it proceeds through all the events (line 4). First, an initial value of the correlation metric is determined: If one element of the pair is present in an event and the other element of the pair is present in the following event, then the correlation metric is assigned the value β_0 multiplied by the probability of each element (lines 6–10). Otherwise, the metric is zero. Second, the metric is adjusted to take into account the category of the next event (lines 11–12). Finally, the metric is adjusted to take into account whether the two elements in the pair are structurally related (lines 22–24). These three steps are then repeated for a comparison of events separated by one event (using the parameter β_1), and then by two events (using β_2).

```
Let $D^* = \{d_1, ..., d_n\} = \bigcup_{i=1}^{m} D_i$
for $i = 1$ to $n$ do
    for $j = i + 1$ to $n$ do
        for all $e_k = (c_k, D_k, X_k) \in E$ do
            $m_{i,j} \leftarrow 0$
            if $d_i \in D_k \wedge d_{i,j} \in D_{k+1}$ then
                $m_{i,j} = p(e_k, d_i) \cdot p(e_{k+1}, d_{i,j}) \cdot \beta_0$
            else if $d_i \in D_{k+1} \wedge d_{i,j} \in D_k$ then
                $m_{i,j} = p(e_{k+1}, d_i) \cdot p(e_k, d_{i,j}) \cdot \beta_0$
            end if
            if $c_{k+1} =$ C then
                $m_{i,j} = m_{i,j} \cdot \beta_C$
            else if $c_{k+1} =$ R then
                $m_{i,j} = m_{i,j} \cdot \beta_R$
            else if $c_{k+1} =$ L then
                $m_{i,j} = m_{i,j} \cdot \beta_L$
            else if $c_{k+1} =$ K then
                $m_{i,j} = m_{i,j} \cdot \beta_K$
            else if $c_{k+1} =$ S then
                $m_{i,j} = m_{i,j} \cdot \beta_S$
            end if
            if $related(d_i, d_{i,j})$ then
                $m_{i,j} = m_{i_j} \cdot \beta_S$
            end if
            {Repeat with $k$ and $k + 2$, using $\beta_1$.}
            {Repeat with $k$ and $k + 3$, using $\beta_2$.}
        end for
    end for
end for
```

Figure 3. Calculating correlation metrics

4.3. Generating Concerns

Once all the pairs have an associated correlation metric, we can generate concerns. The concern generation phase of the algorithm is parameterized in terms of the approximate number of elements desired in all of the concerns reported by the algorithm (η). To generate concerns for a number of elements η, we list pairs of elements generated in the previous phase in decreasing value of m until the number of different elements in all of the pairs is equal to η (or $\eta + 1$). Finally, we group the elements into clusters by taking the transitive closure of every relation represented by a pair in the set of selected pairs. For example, let us assume that for a certain transcript, parameters, and *related* function, $\eta = 5$ yields the following pairs: [A,B][B,C][D,E]. In this case, the algorithm would produce two concerns: [A,B,C], and [D,E]. Once a list of concern descriptions is produced, a developer can choose which descriptions represent the implementation of actual and useful concerns considered during the program investigation, and name and save the useful descriptions for later use.

5. Empirical Evaluation

We have investigated the usefulness and accuracy of our algorithm using data from two replicated studies of program evolution. In both studies, developers were asked to investigate a program in the context of an evolution task using the Eclipse platform, an integrated development environment for Java [6]. For each study, we have analyzed a transcript of the program investigation and have produced a list of concerns with different configurations of the algorithm parameters. This section describes the state of our implementation of the support for concern inference, describes the different parameter configurations we have tried, briefly describes the studies from which we have collected the data, and discusses the results of our investigation.

5.1. Implementation Status

To obtain the results described in this paper, we have generated the program investigation transcripts manually, based on a movie of the the screen recorded during the studies using screen capturing software at full resolution. Although this approach is suitable for the evaluation of the algorithm, use of our approach will require this step to be automated. It should be possible to automate the production of investigation transcripts with appropriate instrumentation of the Eclipse platform. We implemented the concern inference algorithm in Java. To provide the *related* function, we created databases of relations for each case using the bytecode analyses of the FEAT tool (version 2.1.8) [7].

5.2. Configurations

Based on a combination of intuition and experimentation, we have designed five parameter configurations for the concern inference algorithm intended to emphasize different styles of investigation. In general, we found that the algorithm was fairly stable. All parameters require a minimum variation in the order of 10^{-1} (and often in the order of 10^0) to affect a change to the result. The configurations we considered are the following (see Table 1 for the corresponding parameter values):

1. **Basic** A configuration based on our intuition of what should be clues to important elements in the program navigation. Essentially, linear progression based on closeness in the event sequence, more weight on structural, browser, and keyword events, and less on recall and local.

2. **Neighbors** A configuration only taking into account events directly succeeding each other. That is, with parameters $\beta_1 = 0$, and $\beta_2 = 0$.

Table 1. Configuration parameter values

C.	β_0	β_1	β_2	β_B	β_C	β_R	β_L	β_K	β_S
1	3	2	1	1.3	1.5	1.1	0.1	1.4	1.5
2	3	0	0	1.3	1.5	1.1	0.1	1.4	1.5
3	3	2	1	1.3	1.5	1.1	0.1	1.4	1.0
4	3	2	1	1.5	2.0	0.5	0.0	1.5	2.0
5	3	2	1	1.3	1.3	1.0	0.3	1.3	1.2

3. **No Structure** A configuration only taking into account actions of the developer, ignoring underlying structure (i.e., $\beta_S = 1$).

4. **Structure** A configuration putting emphasis on transitions motivated by structural hints.

5. **Guesses** A configuration putting relatively more weight on guessing and browsing.

5.3. Studies

The first set of data is taken from a previous study of program evolution [8]. This study involved programmers making a change to the jEdit text editor.[1] jEdit consists of approximately 65 000 non-comment, non-blank lines of source code, distributed in 20 packages. In the study, subjects were asked to enhance a feature of jEdit pertaining to the automatic backup of unsaved buffers. Before making the change, the subjects were asked to investigate the code of jEdit for one hour and to take notes as necessary. During this time they were not allowed to modify the code or run the debugger. The subjects were also provided with clues consisting of two classes relevant to the change. After the program investigation phase, the subjects were asked to implement the change. From this study we use data from three of the subjects, coded C2-C4 (in reference to [8]). By studying how each subject performed the change, we could determine four important pieces of information about the source code that needed to be considered during the task:

- **Recovery:** A method call performed to recover from an auto-save backup file.
- **Timer interval:** A method call to change the interval of the auto-save timer.
- **Auto-saving:** A method call to save a file buffer in response to an auto-save timer event.
- **Buffer management:** The accesses to a field representing the auto-save backup file.

[1]Version 4.6-pre6, http://www.jedit.org.

To evaluate the results of our algorithm on a different task, we performed another program investigation study. In this second study, we asked two developers to investigate how they would improve a weakness in the implementation of jHotDraw[2], a Java drawing application consisting of approximately 14 600 non-comment, non-blank lines of code distributed in 11 packages. The change posited in this study regarded an incompatibility between commands issued through a menu in the graphical user interface and the actual commands supported by a figure on the drawing canvas. In this study, the subjects were asked to investigate the code of jHotDraw for 45 minutes to plan how they would execute the change. As opposed to the jEdit study, the subjects were not given any initial hint, and were allowed to modify the program to insert print statements. They were not allowed to use the debugger, and were not required to perform the change.

By studying the code of jHotDraw, examining the code investigated by the subjects, and interviewing the subjects, we determined two important pieces of information about the source code that were relevant to the change:

- **Command menus:** A set of methods and classes to build the menus and associate command to each menu item.
- **Figure listeners:** The event-handling system required to detect when the selection of a figure has changed.

In both studies, we recorded all of the activities of the subjects using the Camtesia screen recording program[3] operating at 5 frames/seconds and a resolution of 1280 x 1024 pixels. The resulting movies contained enough information to allows us to produce transcripts as described in Section 3.

5.4. Results

Table 2 describes the size of the transcripts produced by 60 minutes of investigation (subjects C2, C3, and C4) and 45 minutes of investigation (subjects J1 and J2). The second column lists the number of investigation events, and the third column lists the number of different program elements visible to a developer during the investigation.

Using $\alpha = 5$ as our confidence parameter, we applied each of the configurations described in Table 1 to each transcript, requesting in each case the concerns for 12 elements (i.e., $\eta = 12$). For each subject, we present the results in a table. The first column of the table represents an identifier for each concern. The second column presents

[2]Version 5.3, http://www.jhotdraw.org.

[3]http://www.techsmith.com

Table 2. Characteristics of transcripts

Subject	Nb. Events	Nb. Elements
C2	123	71
C3	175	102
C4	204	105
J1	260	200
J2	142	152

Table 3. Results for Subject C2

Id	Concern	1	2	3	4	5
1	A,B	X		X	X	X
2	A,B,C		X			
3	**D,E**	X		X		X
4	**D,E**,M		X			
5	**D,E**,M,P,Q,R				X	
6	F,**G**		X			
7	F,**G,H**	X			X	
8	F,**G,H**,K			X		
9	F,**G,H**,K,L					X
10	**I,J**	X	X	X	X	
11	K,L	X				
12	M,N	X		X		X
13	K,O		X			

Table 4. Results for Subject C3

Id	Concern	1	2	3	4	5
1	**D,E**,M,N,P,R,S,T	X		X		
2	**D,E**,M,N,P,R,S,T,V		X			
3	**D,E**,M,N,R,S,T					X
4	**D,E**,M,P,R,S,T,V				X	
5	**G,H**	X	X	X	X	X
6	F,U	X	X	X	X	
7	K,W,X					X

Table 5. Results for Subject C4

Id	Concern	1	2	3	4	5
1	**I,J**,Q,Y	X	X			
2	**I,J**,K,M,W,X,Y,CC			X		
3	**I,J**,M,Q,Y,BB				X	
4	**I,J**,M,Y,CC					X
5	K,X	X	X			
6	K,W,X					X
7	F,AA		X	X		
8	F,Z,AA	X				
9	F,Z,AA,DD				X	
10	**G,H**	X	X	X	X	X
11	M,BB	X	X			

the different concerns as sets of elements.[4] The remaining columns list the five configurations: an X indicates that the concern denoted by the row was produced for that configuration. For each subject, alternative descriptions of a single user-level concern are grouped together and separated by double lines. To simplify the presentation of the results, we have converted our element codes into sequential letters (for each study, a code represents the same element between subjects). For each subject, we discuss the results based on three evaluation criteria: variability in the number of concerns, variability in the number of elements identified, and relevance of the concerns. We give a general comparison of the data between subjects in Section 5.5.

For subject C2 (Table 3) applying the five configurations produced 13 different concerns involving 18 different elements. Within the 13 concerns generated, three of the important pieces of information described in section 5.3 were identified: **recovery** (D,E, in concerns 3,4,5), **timer interval** (G,H, in concerns 7,8,9), and **auto-saving** (I,J, in concern 10). Other elements are, to varying degrees, less relevant and would probably not be worth saving as a concern description. The important relations were identified by most of the configurations. The most successful configuration in this case was 1 (basic), closely followed by 3 (no structure). This means the subject naturally navigated along the structure, so that existing relations did not need to be factored in.

For subject C3 (Table 4) the five configurations produced more homogeneous results than C2: 7 different concerns involving 16 different elements. Moreover, concerns 1 to 4 are essentially the same concern, with a variation of one or two elements. This concern represents the interaction **buffer management**. Variations on this concern capture how an auto-save backup file is deleted and the various situations in which it is deleted. It is a useful concern, which integrates the interaction **recovery** (D,E). Of the four concerns (1-4), concern 1 is the most accurate. It is present in configurations 1 (basic) and 3 (no structure). Other concerns generated for this subject include the important interaction **timer interval** (G,H, concern 5, present in all five configurations). Concern 6 is spurious, and concern 7 represents the three methods of the class provided as a starting point for the task. In the case of subject C3, the most useful configurations are 1 (basic) and 3 (no structure), as in the case of C2.

For subject C4 (Table 5) the five configurations produced 11 different concerns involving 15 different elements. Concerns 1 to 4 capture the interaction **auto-saving** (I,J). Concerns 5 and 6 list some of the methods of a class

[4]Because the algorithm selects pairs of elements, as opposed to single elements, some parameter configurations resulted in 13 elements being identified.

Table 6. Results for Subject J1

Id	Concern	1	2	3	4	5
1	A,B,C	X	X	X	X	X
2	D,E	X		X		
3	D,E,F,G		X			
4	D,E,F,G,O				X	
5	D,E,F,G,P,Q					X
6	F,G	X				
7	F,G,M			X		
8	H,I	X		X		
9	H,I,J					X
10	J,K			X	X	
11	J,K,L	X	X			
12	M,N		X		X	

Table 7. Results for Subject J2

Id	Concern	1	2	3	4	5
1	R,S,T,U,V,W,X,Y,Z	X				
2	R,S,T,U,V,W,Z,FF,HH					X
3	R,S,T,U,W,X,Y,Z,DD,EE,FF,GG		X			
4	R,S,T,U,W,X,Z,FF,II				X	
5	R,T,U,V,W,Y,Z,FF,HH			X		
6	AA,BB,CC	X		X	X	

used as a hint. Concern 10, identified in all configurations, is exactly the interaction **timer interval** (G,H). All the other concerns are not useful. Given this assessment, configurations 1 (basic), and 2 (neighbors) yield the results that would be most likely to be useful, although the distinction is not as sharp as in the case of C2 and C3.

In the case of the jHotDraw study, for subject J1 (Table 6) the five configurations produced 12 different concerns involving 17 different elements. Concern 1 is a subset of the interactions relevant to the concept **command menu** identified in Section 5.3. Concerns 2 to 7 include different elements related to the construction of the application's menu bar, with the most accurate being concern 5. The other concerns cannot be considered helpful information. In this case, configuration 5 (guesses) yields the most useful concerns.

Finally, for subject J2 (Table 7) the concern inference algorithm produced six different concerns involving 13 different elements. Concerns 1 to 5 are essentially small variations on one major set of elements, which mostly represents interactions implementing the concern **figure listeners**. Concerns 1 and 4 are equally accurate, with six relevant elements out of nine. These correspond to configurations 1 (basic) and 4 (structure). Concern 6 can be considered spurious.

5.5. Discussion

Besides helping us assess the feasibility of inferring concerns automatically from program investigation activities, this study allowed us to make several observations. We discuss these observations and how we plan to move forward on this research.

Successful configurations

In most cases (C2, C3, C4, and J2), configuration 1 (basic) yielded the most useful results. This follows our intuition that transitions between elements in the source code based on browser selection, cross-references, and keyword searches are more important than transitions uncovering elements by scrolling or recalling previous views. In two cases (C2 and C3), configuration 3 (no structure) also yielded good results. Configuration 3 adds no additional weight to a sequence of investigation involving two elements directly related in the code. One possible explanation for the fact that this configuration was successful for C2 and C3 is that both of these subjects were very organized in their program investigation, investigating elements that were related in the first place [8]. In the case of J1, configuration 5 (guesses) was the most successful. This agrees with the behavior of J1, who mostly read source code by browsing up and down the declaration of classes matching general regular expressions. The case of J1 was the least successful application of our algorithm.

Effects of scrolling

Given the nature of the transcripts we use, scrolling a file while investigating code has a drastic effect on the number of events generated. When scrolling, the set of elements visible in an editor window can change as often as multiple times per second. If an element is visible in many of such events, there is a risk that this element will be selected as relevant on the basis that it is involved in many transitions. Our algorithm deals with this situations in two ways. First, an element revealed through browsing does not have a high associated probability (see Section 4.1). Second, the effect of browsing can be mitigated through a low value of β_L. For example, with $\beta_L = 0.1$ and $\beta_B = 1.0$, an element would have to be present in 10 local events before becoming more important than an element revealed a single time through a browser access.

Transcript boundaries

The setting of the jHotDraw study had a few differences with the jEdit study. An important one is that subjects in the jHotDraw study were not given any hints about where to start investigating the code. This resulted in a much broader search for both subjects. This observation is reinforced by the fact that no elements identified in the concerns for J1 overlapped with the ones identified for J2. In contrast, the concerns generated for subjects C2, C3, and C4 were much more focused, and useful, than the ones generated for J1 and J2. These observations seem to indicate that not all of the span of a program investigation session should be used to infer concerns. This raises the important question of when should a program investigation transcript begin and end. Ideally, a developer should be able to deactivate transcript recording when performing broad searches, or while "being stuck", and reactivate the recording when performing more productive investigation. The resulting, more focused, transcripts should yield more accurate concerns.

False positives

As expected, every application of the algorithm resulted in some false positives (or spurious concerns) being generated. This is expected given the nature of the data analyzed. However, anecdotally, we have found that for $\eta = 12$, the number of concerns is low; examining and rejecting false positives in this case is not effort-consuming. Although we do not know if this result will generalize, we do not expect the effort to be significant enough to detract users from using the technique given the potential benefits.

Future work

We envision the complete integration of the approach into a development environment that would allow users to choose between different configurations before generating concerns. This will require research into the optimization of certain configurations for certain investigation styles. Alternatively, it might be possible to add a phase to the technique to automatically detect the best configuration based on general characteristics of the program investigation as can be determined by a cursory examination of the transcript. Finally, we plan to integrate the resulting concerns into FEAT [7], a specialized tool we have developed to represent concerns in source code. This tight integration with FEAT will allow users to immediately see the structural relations between the different elements in the concerns produced by the algorithm and to modify and complete the representations identified by the algorithm. The complete and integrated approach should render the documentation of concerns seamless in the program evolution work flow.

6. Related Work

Impact analysis describes a category of analysis techniques aiming to determine the impact of a modification on a software system. Under this banner, a wealth of approaches have been proposed to identify the code implementing one or more features in a system. We discuss here the approaches that automatically produce results at the source code level, and that are closest to our work.

The Software Reconnaissance technique developed by Wilde *et al.* identifies features in source code based on a analysis of the execution of a program [15, 16]. Software Reconnaissance determines the code implementing a feature by comparing a trace of the execution of a program in which a certain feature was activated to one where the feature was not activated. A tool, TraceGraph, was developed to support the visualization of the difference between execution traces, to help identify and locate the code implementing specific features [5]. The technique was applied to various systems, including three industrial systems consisting of between 10 and 28 kLOC of C or C++ code [14]. Results have shown that the technique can be effective to find good starting points for program investigation. However, Software Reconnaissance, like any dynamic analysis approach, depends on the availability and quality of test cases. In contrast, our approach is based on source code, and can be applied to incomplete or incorrect code. More importantly, the features expressible at the user level may not necessarily correspond to concerns a developer wishes to investigate. Often, developers must investigate code overlapping different features to understand enough of the system to respect the existing design. Because it is independent of the execution of specific features, our approach is flexible enough to capture any subset of a program as a concern.

Slicing [11, 13] is an analysis technique that identifies all the statements in a program that can influence the value of a variable at a specific location. As such, slicing can be used in maintenance activities to help find and manage the code related to a specific statement [2]. For the purpose of automatically detecting concerns, one major limitation of slicing is that only one type of concern can be identified: code related through a control- and data-flow criterion. Another drawback of slicing is that it does not discriminate between interesting and boilerplate code. Our approach is intended to address this problem by factoring in a metric of how relevant different methods are based on how and how often they were examined by a developer.

Some design recovery techniques have been proposed to identify code that would constitute a candidate for refactoring into a module or object (e.g., [9, 12]). These approaches are typically based on the analysis of relations between different program elements, such as "x uses y", using clustering algorithms, or concept analysis. In practice, the results of applying these techniques correspond to scattered concerns. However, the resulting concerns are not task-specific: developers cannot infer concerns related to a specific feature. In contrast, our approach allows the identification of smaller concerns, which would typically be a subset of the modules identified by clustering algorithms. Although our approach also partially relies on structural information, it factors in the focus of the developers during program investigation. This allows us to use the algorithm to infer concerns that are of immediate interest to program developers.

7. Conclusion

In this paper, we have described a technique to infer concerns based on the program investigation activities of developers. Our technique integrates elements of static analysis, but its originality lies in its focus on analyzing the source code a developer examines when investigating a program. Our technique can be parameterized to account for different styles of program investigation.

The evaluation of the technique was based on data obtained from five subjects performing two different tasks. We showed that, in every case, at least one relevant concern was identified. Since the amount of information to be generated by our concern inference algorithm is parameterizable, the number of false positive (or spurious concerns) can be adjusted. In our case, we used the algorithm to infer concerns involving 12 program elements. This number resulted in a very manageable level of information. We also observed that the success of the concern inference algorithm seems to be tied to how organized the program investigation activities are: Broad and disorganized searches produce vague and incomplete concerns, while more focused program investigation can yield a high proportion of useful and precise concerns. This situation can be addressed by not recording the program investigation activities during broad investigation.

Using our technique, which we plan to refine and fully integrate in a software development environment, it is possible to easily and rapidly generate descriptions for different concerns developers investigate in source code. These concern descriptions can then be used as supporting documentation during program evolution tasks, as a basis to plan refactorings [1], and potentially to help port a system to an aspect-oriented language [3].

Acknowledgments

This work was funded by an NSERC research grant, a University of British Columbia graduate fellowship, and an IBM Eclipse Innovation Award. The authors are grateful to Alex Brodsky and to the anonymous referees for their useful comments.

References

[1] M. Fowler. *Refactoring—Improving the Design of Existing Code*. Object Technologies Series. Addison-Wesley, 2000.

[2] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, August 1991.

[3] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings of the 11th European Conference on Object-oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, June 1997.

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

[5] K. Lukoit, N. Wilde, S. Stowell, and T. Hennessey. Trace-Graph: Immediate visual location of software features. In *Proceedings of the International Conference on Software Maintenance*, pages 33–39. IEEE Computer Society, October 2000.

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

[7] 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*, pages 406–416, May 2002.

[8] M. P. Robillard and G. C. Murphy. A study of program evolution involving scattered concerns. Technical Report CS-2003-06, Department of Computer Science, University of British Columbia, March 2003.

[9] M. Siff and T. Reps. Identifying modules via concept analysis. *IEEE Transactions on Software Engineering*, 25(6):749–768, November/December 1999.

[10] E. Soloway, J. Pinto, S. Letovsky, D. Littman, and R. Lampert. Designing documentation to compensate for delocalized plans. *Communications of the ACM*, 31(11):1259–1267, November 1988.

[11] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.

[12] A. van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. In *Proceedings of the 21st International Conference on Software Engineering*, pages 246–255. ACM, May 1999.

[13] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

[14] N. Wilde and C. Casey. Early field experiences with the software reconnaissance technique for software comprehension. In *Proceedings of the International Conference on Software Maintenance*, pages 312–318. IEEE, 1996.

[15] N. Wilde, J. A. Gomez, T. Gust, and D. Strasburg. Locating user functionality in old code. In *Proceedings of the Conference on Software Maintenance*, pages 200–205. IEEE Computer Society, November 1992.

[16] N. Wilde and M. Scully. Software reconnaissance: Mapping program features to code. *Journal of Software Maintenance: Research and Practice*, 7:49–62, January 1995.