Representing Concerns in Source Code

by

Martin P. Robillard

B.Eng. (Computer Engineering), École Polytechnique de Montréal, 1997

M.Sc. (Computer Science), University of British Columbia, 1999

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

We accept this thesis as conforming
to the required standard

The University of British Columbia

November 2003

Abstract

Many program evolution tasks involve source code that is not modularized as a single unit. Furthermore, the source code relevant to a change task often implements different concerns, or high-level concepts that a developer must consider. Finding and understanding concerns scattered in source code is a difficult task that accounts for a large proportion of the effort of performing program evolution. One possibility to mitigate this problem is to produce textual documentation that describes scattered concerns. However, this approach is impractical because it is costly, and because, as a program evolves, the documentation becomes inconsistent with the source code.

The thesis of this dissertation is that a description of concerns, representing program structures and linked to source code, that can be produced cost-effectively during program investigation activities, can help developers perform software evolution tasks more systematically, and on different versions of a system.

To validate the claims of this thesis, we have developed a model for a structure, called concern graph, that describes concerns in source code in terms of relations between program elements. The model also defines precisely the notion of inconsistency between a concern graph and the corresponding source code, so that it is possible to automatically detect and repair inconsistencies between a description of source code and an actual code base.

To experiment with concern graphs, we have developed a tool, called FEAT, that allows developers to iteratively build concern graphs when investigating source code, to view the code related to a concern, and to perform analyses on a concern representation. Using FEAT, we have evaluated the cost and usefulness of concern graphs in a series of case studies involving the evolution of five systems of different size and style. The results show that concern graphs are inexpensive to create during program investigation, can help developers perform program evolution tasks more systematically, and are robust enough to represent concerns in different versions of a system.

Contents

List of Tables

List of Figures

Acknowledgments

If I completed this thesis and kept most of my sanity, it is only due to the many people who have supported me along the way.

My deepest thanks go to my supervisor, Gail Murphy. Gail has an amazing talent for blending encouragements, motivation, and advice in a subtle and clever mix that not only kept me on the right track, but also made me think I had something to do with it. I must thank Gail for teaching me, in the broadest sense possible, how to do research, and for being an example I will strive to follow for the rest of my career.

Thanks also to my senior colleague Rob Walker for his advice, honest criticism, and LaTeX templates, to Alex Brodsky for reviewing the formalism in the thesis, and to Jason Xu who, through his enthusiasm and hard work during his internship in the Software Practices Lab, has facilitated the evaluation of the tool-related aspects of the thesis.

The Department of Computer Science at UBC was in general a wonderful atmosphere for conducting research. Many thanks go the the numerous students, faculty, and staff members who have contributed to this thesis with their comments, help, and moral support.

I am also thankful to the members of my supervisory and examining committee who have generously contributed their time and expertise: Will Evans, Alan Hu, Jim Little, Panos Nasiopoulos, Raymond Ng, and Barbara Ryder.

Finally, I thank my significant other, Dorothee, who was always understanding and stood with me through the rushes and absences (both mine and hers), and my parents, who have believed in me since day one.

My doctoral studies were financially supported by a Canadian NSERC postgraduate scholarship and a University of British Columbia Graduate Fellowship.

MARTIN P. ROBILLARD

The University of British Columbia
November 2003

Chapter 1

Introduction

Useful programs keep changing. This simple observation, proposed as a law of program evolution dynamics by Belady and Lehman more than 25 years ago [12], is still true today. Although much has changed in the way we design and build programs, the need to repair, adapt, and enhance production software is still a reality for software development organizations [64, 124]. The process of affecting modifications to a software system, often called the maintenance process [128], can vary between, and even within, organizations. Although many models have been proposed to structure the process of software maintenance [16, 79, 80, 117, 153], these models can typically be summarized by the three steps originally described by Boehm: understand the existing software, modify the existing software, and revalidate the modified software [14, 15]. Thus, before performing a modification to a software system, developers must explore the system's source code to find and understand the subset relevant to the change task. The large size of most production software, and the usual pressures on development and maintenance time-frames, render the program exploration activity a serious challenge for developers [17]. These factors make it 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 efficiently discover a sufficient amount of the structure and behavior of a program relevant to a modification. The discovery of this structure and behavior is a difficult task. Besides the basic difficulty of understanding source code [13, 152], developers must usually consider conceptually-related subsets of the structure and behavior of a program addressing specific *concerns*. In this dissertation, we use the term "concern" to refer to any consideration a developer or team of developers might have about the implementation of a program. For example, in a file server application based on the File Transfer Protocol (or FTP server), one possible concern is the requirement to log every file transfer command issued by the client programs. The source code corresponding to this concern might consist of calls to functions such as `log(String)`, and the implementation of these functions.

Unfortunately, it is often the case that the program code corresponding to a concern is not well encapsulated, and ends up being scattered across many modules [69]. For example, in the FTP server application described above, the logging code might be scattered throughout the implementation of all of the modules implementing file transfer commands. Scattered concerns pose an additional challenge to developers, who must reason about which pieces of code interact with which other ones to implement a concern, and about how different concerns interact with each other. The incomplete understanding of a scattered concern prior to a software modification can lead to incorrect or inefficient program modifications [74] or a modification not respecting an existing design [101]. The difficulty of locating and understanding scattered concerns is the first problem motivating the work described in this dissertation. This problem can be stated as follows.

Concern Location and Understanding Problem*: It is difficult for developers to locate and understand the code implementing a concern when this code is not encapsulated within a single module.*

Given the difficulty of locating and understanding the implementation of concerns relevant to a change task, it is desirable to capture, even partially, knowledge about the implementation of a concern. A representation of the knowledge about the implementation of a concern can help developers perform modification tasks by supporting a more systematic investigation of the source code oriented along the lines of concerns, avoiding potentially erratic, "hit-and-miss" code investigation behavior. Additionally, preserving knowledge about the implementation of concerns allows other developers working on tasks involving the same concerns to capitalize on previous effort spent on similar modification tasks. The need to document scattered concerns to support software evolution was previously identified by Soloway et al. [73, 127]. Unfortunately, traditional documentation such as that proposed by Soloway suffers from the two principal drawbacks of any software documentation: it is costly to produce and difficult to maintain consistent with the source code. This observation identifies the second problem this dissertation will address, the problem of concern documentation.

Concern Documentation Problem*: It is difficult for developers to cost-effectively document concerns in source code and to keep the documentation consistent.*

Because of this last problem, concerns are practically never documented, and developers tackling a software change task must usually start their investigation from scratch.

One way to address both of the problems described above is to integrate the production of documentation for concern code with the activity of locating and understanding the implementation of concerns, and by producing a concern description robust enough to be reusable with different versions of a code base. The thesis of this dissertation is that a representation for concerns in source code that can support the task of locating and understanding concern code, and that can represent concerns in more than one version of a program, can help developers evolve programs more systematically.

Thesis: *A description of concerns, representing program structures and linked to source code, that can be produced cost-effectively during program investigation activities, can help developers perform software evolution tasks more systematically, and on different versions of a system.*

In Section 1.1 we describe in more detail the concept of implementation concerns. This section is followed by a case study of program evolution involving scattered concerns (Section 1.2). The case study will serve as a running example motivating the work described in the dissertation. In Section 1.3, we provide a brief overview of tools and techniques that can partially address the problem of concern code location and understanding. In section 1.4, we introduce the representation we propose to address both of the problems identified previously. Finally, Section 1.5 is an overview of the dissertation.

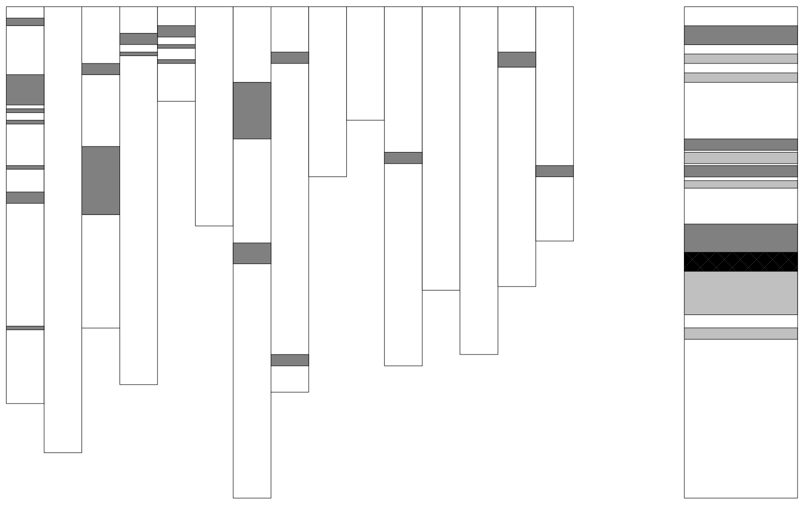

(a) Scattering (b) Tangling

Figure 1.1: Concern scattering and tangling.

1.1 Concerns

The idea of considering separate concerns in the implementation of software originates from Dijkstra [33, 34] and Parnas [99]. From these early works on system design and structured programming, the term “concern” has emerged as a general and flexible notion, intended to include anything a developer might want to consider as a conceptual unit in a program. Examples include the implementation of data stores, algorithms, the need for synchronization, and user interface considerations. Ideally, as a result of the design of a program, concerns should be neatly encapsulated within a module. For example, in a C [66] program requiring the sorting of integer arrays, the “sorting” concern can be encapsulated in a `sort` function. This way, the details of the sorting algorithm, such as whether a quick sort or bubble sort implementation is used, are confined to the `sort` function [122]: modification of the implementation of the sorting algorithm will not require updating the callers of the `sort` function. Unfortunately, in practice, the concerns a developer must consider during program evolution are not always well separated, and their implementation is often found to be scattered through different modules, and, at the same time, tangled within one module [134].[1] Figure 1.1 illustrates schematically the scattering and tangling of concerns. The illustrations use the SeeSoft program view [37], where white rectangles represent the source code for a module (e.g., a C file or a Java [48] class). In the representation of concern scattering *(a)*, gray rectangles indicate source code relevant to a concern. This source code is scattered across multiple modules. In the representation of tangling *(b)*, we show a single module. The module comprises two concerns, shown by the boxes in the two different shades of gray. These two concerns also have overlapping code, represented by the area in black. In other words, tangling involves the presence of code implementing different concerns within a module.

[1]Alternatively, one may say that the concerns *crosscut* the basic program decomposition [69].

The scattering and tangling of concerns in source code is the consequence of four principal causes: inadequate design, the fundamental limitations of programming languages, emergence during program evolution, and code decay.

Inadequate Design Sometimes, scattered concerns result from a failure on the part of the initial developers of a system to create modules hiding implementation details associated with a concern [99]. The lack of concern separation results in a system that displays signs of scattering and tangling. For example, consider the partial C program of Figure 1.2. Line 2 is the prototype of a function sorting an array of integers. This function takes as its third parameter a value for the size of a buffer used in the sorting of the array. Although the flexibility afforded by the `buffer_size` parameter might help to improve the memory consumption of the function, it has negative consequences for the evolvability of the system. Because details of the sort function implementation are exposed to client code (e.g., lines 78 and 490), a developer asked to improve the sorting algorithm will need to consider this client code as part of the sorting concern. A more evolvable design would have hidden the use of a buffer during sorting within the `sort` function.

```
1:   /* Sort function */
2:   int[] sort( int[] a, int n, int buffer_size );
     ...
78:  sorted_array = sort( current_array, 200, 600 );
     ...
490: sorted_codes = sort( codes, 250, BUFFER_SIZE );
     ...
```

Figure 1.2: Sorting program

In brief, even though guidelines exist to prevent unnecessary coupling between modules, design is a human activity and as such is prone to errors and oversights that result in scattered concerns.

Programming Language Limitations In some cases, competing design and implementation goals make it impossible to separate every concern with only the basic constructs of a programming language. This situation has been called the "tyranny of the dominant decomposition" [134]: a base decomposition of principal concerns is imposed on the system by the designer and the programming language, while secondary concerns must remain scattered. Sometimes, it is possible to mitigate the inflexibility of the dominant modular decomposition through the use of special-purpose design strategies, such as design patterns [46]. For example, the Visitor design pattern [46] is a solution to the problem of separating structure from behavior in hierarchical object collections. Although design patterns can help address a small set of well-identified problems, they do not provide a solution for the majority of idiosyncratic modular decomposition problems. Extensions to popular programming languages have been proposed to help re-modularize scattered concerns into separate modules. These extensions usually come under the banner of *advanced separation of concern mechanisms*. Examples for the Java language include AspectJ [67, 68], and the Hyperspaces approach [97, 134] as embodied in the Hyper/J tool [96]. As we discuss in the rest of this section, there exists causes for concern scattering and tangling besides programming language limitations. While advanced separation of concern mechanisms can provide additional flexibility in separating some concerns, which can lead to less scattering, they cannot address all of the possible causes of concern scattering and tangling. As a result, programs built with advanced separations of concerns mechanisms can also suffer from the presence of scattered concerns [83, 85].

Emergence Another cause for the scattering and tangling of concern code is the *emergence* of concerns. Emerging concerns are concerns that did not exist at one stage of the development of a system, but that do need to be considered as a unit for the purpose of an evolution task. In other words, emerging concerns result from unforeseen changes. Although some flexibility for accommodating emerging concerns can be achieved through the use of design for change paradigms [100] and design patterns [46], not all changes can be foreseen. Furthermore, designing for change inevitably involves an increase in code size and design complexity which can be difficult to justify at development time. For this reason, some organizations are moving towards development practices, such as extreme programming [11], where design for change is avoided and replaced by periodical refactorings [43] of the design and implementation of a system to accommodate emerging concerns. In this latter case, the refactoring of a system is itself a program evolution task, usually requiring the consideration of scattered concerns.

Code Decay The last cause for the presence of scattered concerns in source code that we discuss is code decay. This phenomenon can be described with Lehman and Belady's second law of program evolution dynamics, the *Law of increasing entropy*.

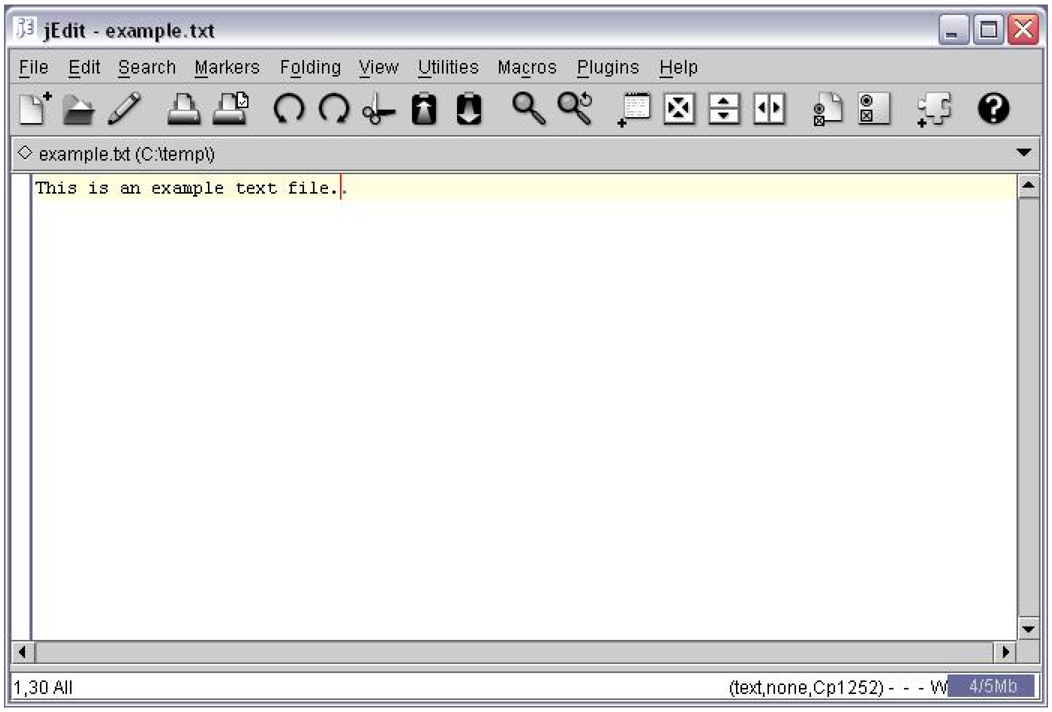
> The entropy of a system (its un-structuredness) increases with time, unless specific work is executed to maintain or reduce it [71: p.169]

Strictly speaking, as digital media, programs are not altered by the sole effect of time. The real cause for the decay of programs is repeated maintenance [36]. The difficulty of locating and understanding the code relevant to a change, the absence of design documentation, the lack of adequate techniques for determining the impact of a modification [18], and time pressure all contribute to software modifications being performed by developers lacking a complete understanding of the implementation of the relevant concerns. As a result of such "ignorant surgery" [101], design constraints are violated and additional coupling between modules is introduced, often resulting in a further scattering and tangling of concerns in source code.

1.2 An Example of Program Evolution Involving Scattered Concerns

We illustrate the problem of locating, understanding, and documenting concerns during program evolution with an example of feature enhancement in a medium-size open-source project. The system we use for this example is the jEdit text editor.[2] The jEdit application is written in Java and consists of approximately 65 000 non-comment, non-blank lines of source code, distributed over 301 classes in 20 packages. jEdit allows users to view and edit text files (called *buffers*), perform regular expression searches, etc. Figure 1.3 show the main window of jEdit. Among its many features, jEdit saves open file buffers automatically. Our example 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 (see Figure 1.4). If jEdit

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


jEdit - example.txt
File Edit Search Markers Folding View Utilities Macros Plugins Help
example.txt (C:\temp\)
1,30 All
(text,none,Cp1252) - - - W 4/5Mb

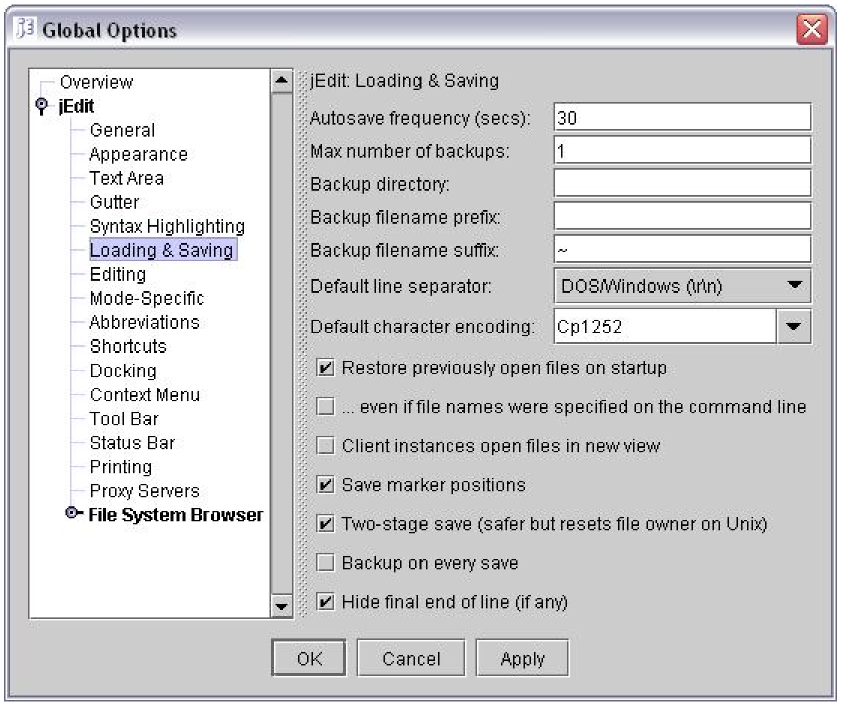
```
This is an example text file.
```

Figure 1.3: The main window of the jEdit application

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. However, this option is undocumented, and can only be discovered by inspecting the source code.

Let us assume the following modification request for the jEdit program is assigned to a developer.

> The application should be modified so that 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 shall 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 shall be saved within the specified autosave frequency.
> 5. When the autosave feature is disabled, the tool shall not attempt to recover from an autosave backup, if for some reason an autosave backup is present. In this case the autosave backup shall be left as is.



jEdit: Loading & Saving

Autosave frequency (secs): 30
Max number of backups: 1
Backup directory:
Backup filename prefix:
Backup filename suffix: ~
Default line separator: DOS/Windows (\r\n)
Default character encoding: Cp1252

[✓] Restore previously open files on startup
[ ] ... even if file names were specified on the command line
[ ] Client instances open files in new view
[✓] Save marker positions
[✓] Two-stage save (safer but resets file owner on Unix)
[ ] Backup on every save
[✓] Hide final end of line (if any)

Figure 1.4: The options window of the jEdit application

Executing this modification request requires understanding different, scattered, implementation concerns. At first glance, without investigating the code, we can already identify potential concerns, such as the implementation of widgets on the options pane, the timing of the autosave event, the management of the buffer state (dirty or not), and the implementation of the autosave recovery operation.

```
public void _init()
{
   /* Autosave interval */
   autosave = new JTextField(jEdit.getProperty("autosave"));
   addComponent(jEdit.getProperty("options.loadsave.autosave"),autosave);
   ...
}

public void _save()
{
   Edit.setProperty("autosave",autosave.getText());
   ...
}
```

Figure 1.5: Partial code in the file `LoadSaveOptionPane.java`

Let us first look at some of the code managing the options pane, as shown in Figure 1.5. The code partially shows two methods of the class `LoadSaveOptionPane`: `_init()` and `_save()`. Looking at this code, a developer can easily determine that the `_init()` method is responsible for creating the text field used for the input of the autosave frequency. Examining the rest of the method (not shown here), it would be possible to establish that all the code for creating the widgets for the option pane is located in the `_init()` method. The WIDGET CREATION concern is thus modularized, and the addition of a check box controlling the autosave feature requires the addition of code only to the `_init()` method. Now let us look at the `_save()`, method. Clearly, the method saves the state of a widget somewhere. Let us call this concern SAVING WIDGET STATE. In our scenario, it is necessary to understand this concern because changing the state of the check box controlling the autosave feature must take immediate effect. It is thus necessary to answer several questions:

- When is the `_save()` method called()?
- Where do properties get saved?
- How is the system notified that some properties have changed?
- How is the state of a property accessed?

These are all questions that cannot be answered by simply examining the code of the `_save()` method. The SAVING WIDGET STATE concern is thus scattered. Attempting to answer the first question by eliciting the call sites for the `_save()` method reveals that it is called at a single point, within method `save()` of class `AbstractOptionPane`, the superclass of `LoadSaveOptionPane`. The definition of the `save()` method is shown in Figure 1.6.

```
public void save()
{
   if(initialized)
      _save();
}
```

Figure 1.6: Method `AbstractOptionPane.save()`

Far from elucidating the circumstances in which the `_save()` is called, the identification of the call site reveals additional complexity for SAVING WIDGET STATE. First, one now must determine the circumstances in which the `save()` method of class `AbstractOptionPane` is called, and the circumstances in which the `initialized` field is true. Since `LoadSaveOptionPane` is a subclass of `AbstractOptionPane`, additional investigation might also be required to determine whether methods of `LoadSaveOptionPane` can affect the state of the `initialized` field. Taking an additional step in the investigation of SAVING WIDGET STATE, we show the source code of method `OptionGroup.save()`, the method calling `AbstractOptionPane.save()` (Figure 1.7). In this case we see that the call to the `save` method (line 12) is embedded in some structure traversal code.

Further investigation would show that detecting when the `_save()` method is called requires eliciting relationships between at least nine methods and one field in six different classes. Reasoning about all these interactions at once is impeded by the effects of both scattering and tangling. Because the concern is scattered, using an integrated development environment, the developer must understand and reason about the implementation of the concern by cycling through multiple editor windows, each providing only a fragment of the information required to understand the concern. Because the implementation of the concern is tangled, each piece of code implementing the concern is cluttered with details not pertaining to the concern, such as structure traversal or error handling code (as exemplified in Figure 1.7).

```
public void save()
{
   Enumeration enum = members.elements();

   while (enum.hasMoreElements())
   {
      Object elem = enum.nextElement();
      try
      {
          if (elem instanceof OptionPane)
         {
            ((OptionPane)elem).save();
         }
         else if (elem instanceof OptionGroup)
         {
            ((OptionGroup)elem).save();
         }
      }
      catch(Throwable t)
     {
         Log.log(Log.ERROR, elem,
                 "Error saving option pane");
         Log.log(Log.ERROR, elem, t);
      }
   }
}
```

Figure 1.7: Method `OptionGroup.save()`

Going back to the four questions posed about the implementation of SAVING WIDGET STATE, we observe that three of them relate to the management of a collection of property objects. This raises the additional question of whether properties are used only to store the state of the widgets, or are used as a form of global variables to store properties of jEdit. In the latter case, it would be important to gather a minimum understanding of how properties work to ensure that any modification involving properties management respects the existing design. As such, it might be useful to consider global properties management as a separate concern interacting with SAVING WIDGET STATE. The presence of properties management code within SAVING WIDGET STATE illustrates the important point that concerns do not exist in isolation. Concerns are integrated in an existing code base, they interact with other concerns, and the boundaries between different concerns and between a concern and the base code is not clearly defined. Reasoning about fuzzy boundaries for concerns adds an additional dimension to the difficulty of dealing with scattered concerns. In fact, studies have show that interacting concerns are often construed as major obstacles during program evolution tasks [8].

To summarize, a program evolution task such as the enhancement of the autosave feature in jEdit requires considering different concerns, such as saving the information contained in user interface widgets, and managing global properties of the application. Although concerns can often be simple and obvious concepts at an abstract level, their implementation is often scattered and tangled, making it difficult to fully understand their structure and behavior at once. Additionally, the boundaries of concerns are not clearly defined, and concerns often interact with other concerns, making it difficult to focus on a single concern at the time.

1.3 Existing Support for Scattered Concerns

Many program understanding and reverse engineering approaches have been developed to help a developer discover the code related to a maintenance task. In this section we present a brief overview of the different types of approaches currently available to developers. Chapter 7 provides a more comprehensive survey of techniques that can help developers find and manage the code implementing scattered concerns.

Searching and cross-referencing Lexical searching tools, such as grep [2], and cross-reference databases, such as the C Information Abstractor [25], can help a developer identify points in the code relevant to a concern, and, in the case of cross-reference databases, the relationships between the different points identified. Regular expression searches and cross-reference queries have also been integrated in software development environments [47, 93, 94, 119]. However, the basic searching and cross-referencing facilities of integrated environments follow a discover-and-discard model that provides little or no help for managing, understanding, and preserving the information discovered. At best, query results will be kept in a history list, allowing developers to revisit the results of searches. We argue that such minimalistic features cannot properly help developers document concerns in large, evolving code bases because they do not help the developer localize and synthesize the information related to the concern of interest [7]. Additionally, basic search mechanisms do not provide a means to document concerns across versions of a system. In brief, basic search features are not intended to, and do not, address the concern documentation problem.

Program Slicing Program slicing denotes a type of analysis intended to identify the parts of a program that may affect the values computed at some point of interest [138]. Slicing was originally defined as a static analysis technique [146], but many variants have since been proposed, including variants relying on dynamic information. Slicing and similar techniques can be used in maintenance activities to help find and manage the code related to a specific statement [45]. From the perspective of finding and understanding concerns, a 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 that would typically be ignored by developers (e.g., calls to a low-level library).

Reverse engineering and design recovery Reverse engineering [26] tools provide developers with views of the different elements in a program (e.g., classes, methods), and of the relations between them (e.g., Rigi [81]). Such tools can be construed as a visual representation of cross-reference tools. As such, they do not address the concern documentation problem.

Revision history A developer wishing to determine which subsets of the source code were affected by previous modifications can rely on data from a version control system such as RCS [137], SCCS [116], or CVS [22]. Although source code identified in this fashion can help point to interesting concerns, it often represents an incomplete picture of a concern. For instance, concerns that a developer needs to understand typically involve much more source code and program interactions than the code that was actually changed [8].

Dynamic analysis Other approaches to finding code relevant to one or more concerns use information about a program's execution. For example, using the Software Reconnaissance technique [149, 150], a developer 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. Software Reconnaissance and other dynamic analysis approaches like it, however, depend on an available suite of quality test cases. 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 existing design.

Concern analysis Finally, specialized program navigation and analysis tools have been proposed that to address the problem of scattered concerns. For example, conceptual Modules [7] 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 [50] allows users to find and assess concerns based on which lines of code match user-specified regular expressions. Most of these tool address a very specific issue relating to scattered concerns, be it discovery and visualization (AspectBrowser), or analysis (Conceptual Modules). None of the approaches proposed focus on documenting concern representations as a general, long-term artifact for program evolution.

1.4 Concern Graphs

To help developers locate, analyze, understand, and document scattered concerns during multiple evolution tasks on a program, we propose an approach that relies on a representation of concerns in source code as a principal artifact. We call our concern representation a *concern graph*. Simply put, a concern graph is a description of a subset of a program relevant to one or more concerns. However, a concern graph representation differs from an actual program in two important ways:

1. it mitigates tangling by abstracting the implementation details of the implementation of a concern, and
2. it mitigates scattering by making explicit the relationships between the different, scattered, pieces of code implementing a concern.

Let us illustrate these properties with the example of Section 1.2. Figure 1.7 shows the code of method `save()` of class `OptionGroup`. The subset of this method relevant to the SAVING WIDGET STATE concern is the call to method `save()` of class `AbstractOptionPane` (line 12). The statement corresponding to this fact is reproduced here:

```
((OptionPane)elem).save();
```

Although simple, this statement contains information that is extraneous to our concern investigation. Specifically, in our case, it is not necessary to know that the `save()` method is called on the object stored in the `elem` variable. Moreover, we do not need to know that a down cast is required for this call to correctly type-check. The mere presence of this method call contributes to our understanding of the concern, and additional details resulting from the tangling of the call with the traversal of an enumeration is mere cluttering. Recognizing this problem, our intent for the concern graph representation is to do away with the useless details, and store only the essential information, that `OptionGroup.save()` calls `AbstractOptionPane.save()`.

The second difference between a concern graph and the source code is that a concern graph documents explicitly the relationships between the different elements of a concern. Returning to our example, we see that the method call statement shown above does not tell us where the implementation of the `save` method exists. Although the down cast to `OptionPane` tells us that is will be found in a class implementing or extending `OptionPane`, cross-reference searches are required to obtain this information. In contrast, a concern graph stores this information explicitly. To complete our example, the information stored in a concern graph for the statement above includes:

```
OptionPane.save() CALLS AbstractOptionPane.save()
```

Of course, a concern graph is not limited to a *single* interaction between two elements (e.g., classes, methods, fields, variables, constants, etc.), but can comprise many different interactions, together forming a graph of relationships corresponding to the implementation of a concern. For example, Figure 1.8 shows the graph corresponding to the information about jEdit elicited in Section 1.2.

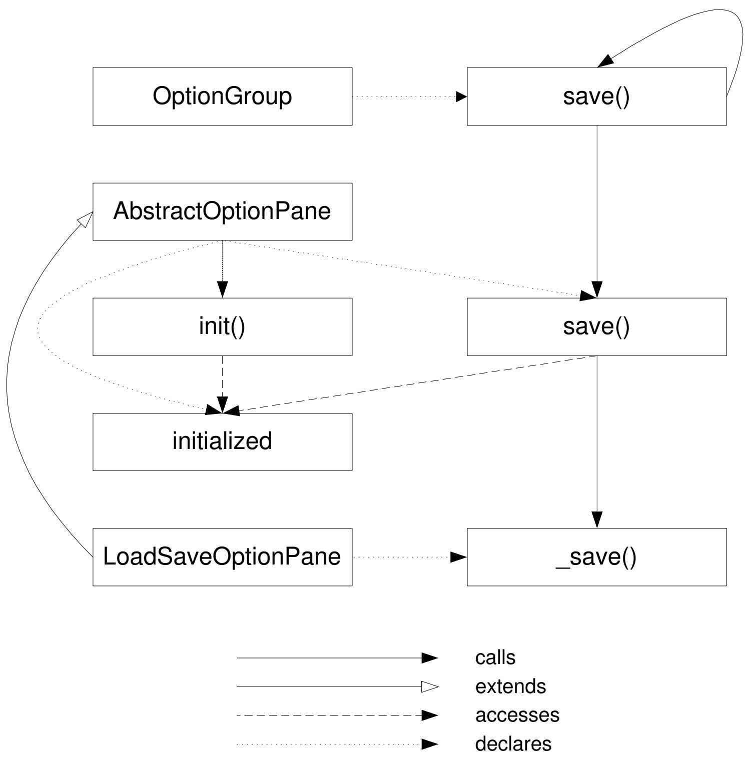


Figure 1.8: Partial concern graph for SAVING WIDGET STATE

In the figure, nodes represent elements declared in jEdit (three methods, one field, and three classes), and edges represent the relationships between the elements. The graph provides a single and abstract view of the implementation of (part of) the SAVING WIDGET STATE concern, enabling a developer to reason about only the code of interest.

A concern graph must be based on a program model that can be extracted automatically from either the source code or an intermediate representation of a program. As a result, a developer is able

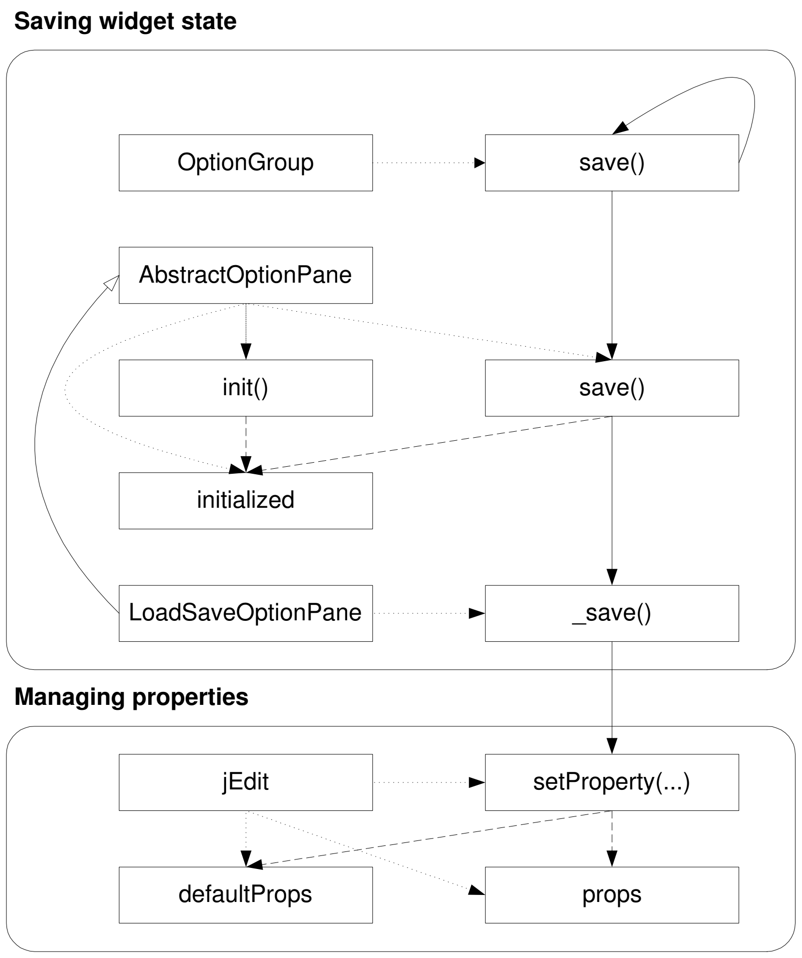


Figure 1.9: Partial concern graph for SAVING WIDGET STATE and MANAGING PROPERTIES

to manipulate and navigate a concern representation at a more abstract level than the source code without investing any effort to create the abstract representation. Automatically providing the part of a program model related to an element allows a concern graph to be augmented incrementally from related elements in the code base, potentially to include more than one concern. For example, we can augment the graph of Figure 1.8 to include some of the elements related to the management of properties (shown in Figure 1.9). The concern graph allows us to investigate the interactions between the two concerns. Of course, SAVING WIDGET STATE and MANAGING PROPERTIES can also be specified as two different concern graphs, and merged at a later stage.

Finally, since concern graphs are intended to represent the source code relevant to a concern, there should exist an unambiguous mapping between the abstract representation and the source code. In other words, it should not be possible to specify information as part of a concern graph that cannot be automatically and unambiguously associated with source code. Inspection of the graphs in Figures 1.8 and 1.9 demonstrates that all of the structures present in the graph can be mapped back to the original code in jEdit: the nodes correspond to the declarations of classes, methods, and fields, and the edges correspond to more specific, sub-method information, such as method calls.

To summarize, the approach we propose to address the problems of concern location and understanding and the problem of concern documentation relies on an abstract representation of concerns in source code called a concern graph. A concern graph describes the structural links between different program elements potentially relevant to a concern, and supports a direct mapping to the corresponding source code.

1.5 Overview of the Dissertation

In the remainder of this dissertation, we present the details of the concern graph approach, demonstrate how eliciting and focusing on different concerns can help a developer perform software evolution tasks more systematically, and demonstrate how the information gathered about the implementation of concerns can be reused with different versions of a system.

In Chapter 2, we present a mathematical model for the definition of concern graphs on a program. This model is general and language-independent, and supports the definition of concerns by combining minimal descriptions, called fragments, into structures of increasing complexity. The concern graph model is also designed to support the detection and repair of inconsistencies between a concern graph and a program, making concern descriptions tolerant to the evolution of the corresponding source code. The description of the concern graph model, and of the mechanism we designed to detect and repair inconsistencies between a concern graph and the source code, form the first two contributions of this dissertation.

To evaluate the practical value of representing concerns in source code, we instantiated our concern graph model for the Java language. In Chapter 3, we provide the details of the concrete concern graph model we produced for use with Java programs, and we discuss the issues of usability and scalability related to our concrete model for Java. The presentation of the Java model and the accompanying discussion constitutes the third contribution of this dissertation. In Chapter 3, we also describe the tool we implemented to support the use of concern graphs with Java programs. This tool, and the discussion of the challenges we addressed regarding its design and implementation, form the fourth contribution of this dissertation.

Based on the tool support for concern graphs described in Chapter 3 we conducted five case studies to validate the thesis of this dissertation. In Chapter 4, we describe each case study to show that concern graphs can help developers perform evolution tasks more systematically, are inexpensive to create, and are robust enough to be used on different versions of a system. The description of the design of the five case studies, and the discussion of the problems we have encountered, and of the steps we have taken to address them, constitutes the fifth contribution of our work.

The basic concern graph approach, as introduced in this chapter, relies on developers manually building concern graphs during program investigation activities. However, one way to further reduce the cost of producing concern descriptions is through automation techniques. In Chapter 5, we show how we can lower even further the cost of producing concern graphs by describing an algorithm for automatically inferring concern descriptions from the program investigation activities of a developer. The algorithm presented in Chapter 5 is the sixth contribution of this dissertation.

In Chapter 6, we describe the main issues that arose during the development and investigation of the concern graph approach, summarize our views on the potential impact of concern graphs on the process of software evolution, and present a plan for future research involving concern graphs.

In Chapter 7, we put our research in perspective and highlight its novelty by providing an overview of the related work. Finally, in Chapter 8, we conclude and summarize the contributions of the work described in this dissertation.

Chapter 2

The Concern Graph Model

Generally speaking, a *concern* is any consideration a developer or team of developers might have about the implementation of a subset of a program. A *concern graph* is an artifact intended to describe the source code that might be relevant to a concern. A concern graph is thus associated with a program. To explicitly state what can and cannot be expressed about a program in a concern graph, and the type of reasoning and analyses that developers can perform on concern graphs, we define a formal model of concern graphs.

2.1 Design Goals

The concern graph model is designed to meet several goals: to be language independent, flexible, precise, simple, robust, and tolerant to inconsistencies.

The first requirement is for the concern graph model to be *language independent*. The concept of an implementation concern is by no means limited to a particular programming language. Although mapping a concern to source code must inevitably involve the consideration of programming language syntax and semantics at some level, we wanted the general structure representing concerns to be language independent, to enable any reasoning performed at the level of the model to be applicable to concerns for code in any language.

The requirement for flexibility relates to the type of information that can be expressed about a program. For example, to capture concerns about the hierarchical organization of modules in a program, it is necessary only to capture information about the interactions between high-level declarations, such as classes or data structures. To capture concerns about general control-flow of an imperative program, it is necessary to represent interactions (calls) between methods or functions. Finally, to capture concerns about the flow of data during the execution of a program, it is necessary to include local variables in the model. Technically, any information about a program that can be produced statically (by a compiler or specialized analyzer) is available to describe part of a program. Examples include the basic declarative structure of a program, but also control- and data-flow information as represented by the program dependence graph (a program representation used in compiler optimization [40]), or the system dependence graph (used in software engineering applications such as testing [57]). We wanted a model flexible enough to be customizable for different levels of granularity of program information.

Our third requirement for a concern graph model is for it to be *precise*. Here, by precision, we mean that there should exist a non-ambiguous mapping between any structure present in a concern representation and the corresponding source code.

As much as possible, we wanted to ensure that the concern graph model remained *simple* and *intuitive*. In this way, developers working with a concern graph can determine the interactions between the different pieces of source code it represents without having to perform complicated calculations or logic reasoning.

The last two requirements, *robustness* and *tolerance to inconsistencies*, relate to the capability of concern graphs to represent concerns in evolving source code. Since a concern graph is essentially a representation of a subset of a code base, changes to this code base are bound to affect the representation. The requirement for robustness states that a concern graph should remain valid through minor code modifications. As such, it should not be dependent on non-essential and brittle aspects of the source code, such as line numbers or indentation. Also, major source code modifications affecting the code represented by a concern graph should not invalidate the concern graph. Rather, it should be possible to detect any inconsistencies between a concern graph and its associated code base and to use the consistent part of a concern graph, while preserving the inconsistent information to help repair the inconsistencies.

2.2 Formal Representation

We define concern graphs formally as a mathematical model based on relational algebra. Appendix A presents the notation and important relational operations we use in the definition of the model.

2.2.1 Programs

Concern graphs must be able to represent a subset of a program that relates to a concern a developer has about the implementation of the program. As a result, the definition of a concern graph must be linked to an underlying program model that specifies which information about a program can be captured by a concern graph. This section deals with the modeling of programs. It forms the groundwork on which the concern graph model is built.

Our model of a program relies on the notion of a *named relation*. Named relations allow us to directly attach a meaning to a mathematical relation.

Definition 1 (Named Relation) *A named relation $R_n = (n, R)$ consists of a name n associated with a binary relation R.*

We model a program as a set of elements declared in the program, and a set of named relations between these elements.

Definition 2 (Program Model) *A program model (E, N) consists of a set of program elements $E = \{e_1, e_2, ..., e_m\}$ and a set of named relations over E, $N = \{R_{n_1}, R_{n_2}, ..., R_{n_k}\}$.*

This definition states that anything that can be known about a program in our model must be expressed in terms of relations between elements. The generality of this definition allows the program model to apply to any representation of program code that can be expressed as a set of relations between elements. It applies equally to executable, intermediate, or source code. It applies to stand-alone programs as well as libraries. Finally, it applies to complete and correct programs as well as incomplete or incorrect programs.

For convenience in presentation, we provide a shorthand to represent the set of all relation names in a program model.

Definition 3 (Names Set) *Let $N = \{R_{n_1}, R_{n_2}, ..., R_{n_k}\}$ be a set of named relations. We define the set of all relation names in N as*

$$\text{names}(N) = \{n \mid \exists\, R \,:\, (n, R) \in N\}.$$

This definition of a program model is equivalent to the definition of a labeled directed graph $(E, \text{names}(N), \Delta)$, with E a set of nodes, $\text{names}(N)$ a set of labels, and $\Delta \subseteq E \times \text{names}(N) \times E$ a set of triples representing the labeled edges [102]. The name "concern graph" is thus intended to capture the idea of a graph of elements (nodes) and named relations (labeled edges) representing the subset of a program model addressing a concern.

Given a concrete program P in some programming language, a model of this program is obtained by applying a language-specific *mapping function* M to the program. A model of program P according to mapping function M is represented by P_M. Different mapping functions can be defined for a single programming language.

Definition 4 (Mapping Function) *Let $P_M = (E, N)$ be a program model. The mapping function M consists of:*

- *A criterion defining which elements declared in program P should be listed in E.*
- *A set of relation names supported by the model.*
- *The definition of an analysis function $a(n, P)$ taking as parameters a relation name n and a program P, and returning a named relation $R_n \subseteq E \times E$ representing the relationships between elements of P (meeting the mapping criterion), according to the semantics of n.*

Because mapping functions are a means to obtain a program model from a concrete program, and do not support any reasoning about concern graphs, they will not be further formalized. All the formal reasoning for concern graphs involves modeled programs. In this dissertation, mapping functions are specified in a box, listing from top to bottom:

- the name of the mapping function,
- the criterion for inclusion of an element x of program P into E (the set of elements modeled),
- the set $\text{names}(N)$ of supported relation names, and
- the definition of the analysis function $a(n, P)$.

By convention, analysis functions will be defined here using first-order logic. In practice, other notations can be used. As mentioned above, the definition of mapping functions is only an accessory issue which does not influence the characteristics of the concern graph model.

For example, Figure 2.1 presents a minimal mapping function for the C language modeling a program only as its call graph. The boolean functions used in the definition of the analysis function are normally defined in terms of the language specifications. For the purpose of the simple examples in this chapter, we assume that the behavior of the functions can be inferred from their name. Relation names in a model are set in italics to distinguish them from the names of boolean functions over

Mapping Function C1
$E = \{x \mid \text{IsAFunction}(x)\}$
$\text{names}(N) = \{\mathit{Calls}, \mathit{CalledBy}\}$
$a(\mathit{Calls}, P) = \{(x, y) \vert \text{Calls}(x, y)\}$
$a(\mathit{CalledBy}, P) = a(\text{Calls}, P)^{\top}$

Figure 2.1: Mapping function C1

a program; boolean functions remain in normal type (see Appendix A for the complete notational conventions).

The mapping function C1 states that the only information available about a program in the model is the program's call graph, represented by the relations *Calls* and *CalledBy* (the transpose of the *Calls* relation). At this point it might seem superfluous to specify *CalledBy* as a relation in the mapping function since this relation represents redundant information, which can be obtained by a simple operation on the *Calls* relation. However, there exists an important semantic distinction between the two relations that can provide additional expressiveness in describing concerns. This issue is discussed in further details in Section 2.2.2.

The application of a mapping function M to a program P yields a program model P_M. The application process consists in extracting the concrete set of program elements E_P and named relations N_P.[1] The elicitation is performed by applying the selection criterion to all the elements declared in P, and by applying the analysis function to P for all relation names specified in the mapping function. In practice, this step is performed via standard static analyses, such as parsing [35], type-checking [3], control- or data-flow analysis [84, 91], and exception flow analysis [27, 111, 114, 120, 125].

We illustrate the process of producing a program model with an example of a simple Java program. In Java, in addition to methods and fields, classes can also declare other classes (called *inner classes*). These inner classes can extend any class visible in the scope of their declaration, creating intricate dependencies between different classes. In this example, we define a mapping function for the Java language capable of only representing the declaration and specialization relationships between classes. Figure 2.2 specifies this mapping function.

Mapping Function J1
$E = \{x \mid \text{IsAClass}(x)\}$
$\text{names}(N) = \{\mathit{Declares}, \mathit{Extends}, \mathit{SuperclassOf}, \mathit{SubclassOf}\}$
$a(\mathit{Declares}, P) = \{(x, y) \mid \text{Declares}(x, y)\}$
$a(\mathit{Extends}, P) = \{(x, y) \mid \text{Extends}(x, y)\}$
$a(\mathit{SubclassOf}, P) = a(\mathit{Extends}, P)^{+}$
$a(\mathit{SuperclassOf}, P) = a(\mathit{SubclassOf}, P)^{\top}$

Figure 2.2: Mapping function J1

[1]Variables names for elements in a concrete program model resulting from the application of a mapping function to a program will be indexed with the name of the program. For abstract entities independent from a specific program model, the index is omitted.

This mapping function specifies that the only declarations considered in a modeled program are the classes declared in the program: interfaces, fields, methods, local variables, and other declarations are not modeled. Furthermore, the only relationships between classes documented by this model are whether a class declares another class (*Declares*), whether a class extends another class directly (*Extends*) or transitively (*SuperclassOf*), or whether a class is transitively extended by another class (*SubclassOf*). Like the mapping C1 of Figure 2.1, the mapping J1 illustrates how relations can be defined in terms of other relations to add expressiveness to the model. In this case the use of the transpose and non-reflexive transitive closure operations support the definition of the *SubclassOf* relation. This example also illustrates an additional point: there are no constraints (such as symmetry) on the set of relations specified in a mapping function. This set can be customized to include only relations that offer a useful means of representing a program to users of the model. As such, we have purposefully excluded the transpose of the *Extends* relation as part of the model. If this relation had been deemed to be necessary to describe concerns, it could have easily been added to the model.

Let us now apply this mapping function to the program P1 of Figure 2.3. This application yields the model of Figure 2.4. Program models are presented with a structure similar to mapping functions, but with a double line under the model name.

```
public class A
{
   int aField;
   class B {}
}

class C extends A
{
   void aMethod(){};
   class D extends A {}
   class E extends D {}
}
```

Figure 2.3: Program P1

Model $\mathbf{P1_{J1}}$
$E_{P1} = \{\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}, \mathtt{E}\}$
$\mathit{Declares}_{P1} = \{(\mathtt{A}, \mathtt{B}), (\mathtt{C}, \mathtt{D}), (\mathtt{C}, \mathtt{E})\}$
$\mathit{Extends}_{P1} = \{(\mathtt{C}, \mathtt{A}), (\mathtt{D}, \mathtt{A}), (\mathtt{E}, \mathtt{D})\}$
$\mathit{SubclassOf}_{P1} = \{(\mathtt{C}, \mathtt{A}), (\mathtt{D}, \mathtt{A}), (\mathtt{E}, \mathtt{D}), (\mathtt{E}, \mathtt{A})\}$
$\mathit{SuperclassOf}_{P1} = \{(\mathtt{A}, \mathtt{C}), (\mathtt{A}, \mathtt{D}), (\mathtt{D}, \mathtt{E}), (\mathtt{A}, \mathtt{E})\}$

Figure 2.4: Model $\mathrm{P1_{J1}}$

This application results in the elements $\{\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}, \mathtt{E}\}$ being considered.[2] Even though the program declares other elements, such as field `A.aField` and method `C.aMethod()`, these are

[2]In Java, classes that do not declare to extend any class extend the class `java.lang.Object` by default. For this reason, the `Object` class should be part of E_{P1} and the relations in N_{P1}. We have left this detail out of our example for simplicity.

not included because they do not match the type specification for elements of E according to the mapping function J1. Likewise, only pertinent relations between elements of E_{P1} are produced as a result of the application.

2.2.2 Fragments

With the conceptual foundations for modeling programs in place, we can now address the definition of a concern representation. Simply put, a concern graph is a description of a subset of a program model. Concern graphs are defined as a collection of small building blocks called *fragments*, which can be assembled to form concern descriptions of increasing complexity. A fragment describes a relationship between two sets of elements in a program model. It is the smallest unit of concern graph description. A fragment f_P is always defined on a program model P_M. A fragment consists of an *intent* part and a *program subset* part. The intent of a fragment captures a high-level description of what one wishes to capture about a program (e.g., "all the subclasses of class `C`"). The program subset part captures the actual subset of the program corresponding to the intent (e.g., "classes `A` and `B`").

To define the intent of a fragment, we use a *domain* set, a relation name, and a *range* set.[3] For example, to specify a fragment representing a function call from function `a` to function `b`, we would specify $\{$`a`$\}$ as the domain, $Calls$ as the relation name, and $\{$`b`$\}$ as the range.

To describe the program subset part of a fragment, we need to define a projection operator on the objects defining the intent of a fragment and a program model.

Definition 5 (Projection Operator) *Let $P_M = (E_P, N_P)$ be a program model, $Dom_P \subseteq E_P$ and $Ran_P \subseteq E_P$ be two subsets of E_P, and $n_P \in \mathrm{names}(N_P)$ the name element of a named relation $R_{n,P}$ in N_P.*

$$\mathrm{proj}(Dom_P, n_P, Ran_P, P_M) = Dom_P \lhd R_P \rhd Ran_P.$$

In other words, the projection operator takes the intent of a fragment (a domain, relation name, and range) and a program model, and produces the relation corresponding to the intent. The distinction between the intent and program subset part of a fragment is important when one considers the evolution of programs. A fragment describing code for a program version P1 might still apply to a program version P2. The presence of the projection in the fragment supports answering this question precisely. The algorithms described in Section 2.3.2 detail how inconsistencies between a fragment and a program model can be detected and reconciled through the use of the projection.

We now have all of the tools required to formally define a fragment.

Definition 6 (Fragment) *Let $P_M = (E_P, N_P)$ be a program model. Let $Dom_P \subseteq E_P$ be a domain defined on P, $Ran_P \subseteq E_P$ a range defined on P, and $n_P \in \mathrm{names}(N_P)$. We define a fragment as $f_P = (Dom_P, n_P, Ran_P, Proj_P)$, where $Proj_P = \mathrm{proj}(Dom_P, n_P, Ran_P, P_M)$. We say that f_P is defined on P_M.*

Given these definitions, we see that specifying a fragment consists in specifying a domain and range sets and a relation name, and applying the projection operator on a program model. The resulting

[3]The domain and range of a fragment are set variables, and are not to be confused with the relational operators defined in Appendix A.

fragment describes a subset of the program model. An important consideration when specifying fragments is the specification of the domain and range sets. Technically, for a program model $P_M = (E_P, N_P)$, any specification resulting in a set $S \subseteq E_P$ constitutes a valid domain (or range). In particular, we recognize three types of domain/range specifications:

- A non-empty set of elements (e.g., $\mathit{Dom} = \{\mathtt{A}\}$, $\mathit{Ran} = \{\mathtt{A}, \mathtt{C}, \mathtt{D}\}$).
- The universal domain (or range), represented by the set E_P. Specifying E_P as the domain or range of a fragment will result in the projection including all elements in the domain of the specified relation.
- A subset specified as the range of a fragment projection. For example, to specify a domain as all of the members of class `A` in a program model P_M, we would specify $\mathit{Dom}_P = \mathrm{ran}(\mathrm{proj}(\{\mathtt{A}\}, \mathit{Declares}, E_P, P_M)))$.

This flexibility in fragment specification affords us the capability to capture cohesive groups of relations as one fragment, and allows us to capture the semantic relationships between the relations as the intent of the fragment.

We illustrate different possibilities for fragment specification through a series of examples based on the mapping function J2 for the Java language (Figure 2.5).

Mapping Function J2
$E = \{x \mid \mathrm{IsAType}(x) \vee \mathrm{IsAMethod}(x)\}$
$\mathrm{names}(N) = \{\mathcal{I}, \mathit{Declares}, \mathit{Calls}, \mathit{CalledBy}\}$
$a(\mathit{Declares}, P) = \{(x, y) \mid \mathrm{Declares}(x, y)\}$
$a(\mathit{Calls}, P) = \{(x, y) \mid \mathrm{CallsStatic}(x, y)\}$
$a(\mathit{CalledBy}, P) = a(\mathit{Calls}, P)^\top$

Figure 2.5: Mapping function $J2$

The J2 mapping only considers types (classes and interfaces), and methods. The relationships modeled are restricted to the identity relation ($\mathcal{I}$), the declarative structure of the program, and static method calls. As will be illustrated below, the identity relation serves the special purpose of allowing the definition of fragments corresponding to a single program element.

Based on mapping function J2, we can specify fragments of program P2 (Figure 2.6). The model for program P2 is shown in Figure 2.7.[4]

To describe a single program element as a fragment, we use the identity relation $\mathcal{I}$. In the examples, fragments are named with a phrase describing their intent.

Class A
$(\{\mathtt{A}\}, \mathcal{I}, \{\mathtt{A}\}, \{(\mathtt{A}, \mathtt{A})\})$

We can also describe a single method call as a fragment:

c calls b
$(\{\mathtt{c}\}, \mathit{Calls}, \{\mathtt{b}\}, \{(\mathtt{c}, \mathtt{b})\})$

[4]The notation has been simplified by omitting the parentheses in method signatures.

```
public class A
{
   public static void b(){};
   public static void c(){ c();b(); D.f();}
}

class D
{
   public static void e() { f(); }
   public static void f() {}
}
```

Figure 2.6: Program P2

Model $\mathbf{P2_{J2}}$
$E_{P2} = \{\mathtt{A}, \mathtt{b}, \mathtt{c}, \mathtt{D}, \mathtt{e}, \mathtt{f}\}$
$\mathcal{I}_{P2} = \{(\mathtt{A}, \mathtt{A}), (\mathtt{b}, \mathtt{b}), (\mathtt{c}, \mathtt{c}), (\mathtt{D}, \mathtt{D}), (\mathtt{e}, \mathtt{e}), (\mathtt{f}, \mathtt{f})\}$
$Declares_{P2} = \{(\mathtt{A}, \mathtt{b}), (\mathtt{A}, \mathtt{c}), (\mathtt{D}, \mathtt{e}), (\mathtt{D}, \mathtt{f})\}$
$Calls_{P2} = \{(\mathtt{c}, \mathtt{b}), (\mathtt{c}, \mathtt{c}), (\mathtt{c}, \mathtt{f}), (\mathtt{e}, \mathtt{f})\}$
$CalledBy_{P2} = \{(\mathtt{b}, \mathtt{c}), (\mathtt{c}, \mathtt{c}), (\mathtt{f}, \mathtt{c}), (\mathtt{f}, \mathtt{e})\}$

Figure 2.7: Model $\mathrm{P2_{J2}}$

Fragments containing a single element in the domain and a single element in the range are called *primitive fragments*. If the relation represented by the primitive fragment actually exists in the model, the fragment projection is a set comprising a single pair formed by the single element in the domain and range. If the relation does not exist in the program model, the fragment projection is represented by the empty relation $\mathcal{O}$. As a last example of primitive fragment, we can capture the fact that class `D` declares method `f`:

D declares f
$(\{\mathtt{D}\}, Declares, \{\mathtt{f}\}, \{(\mathtt{D}, \mathtt{f})\})$

Obviously, primitive fragments do not exercise the full expressive power of the fragment structure. We can describe slightly more elaborate interactions using the universal range. For example, to capture all members of class `A`, we specify:

Members of A
$(\{\mathtt{A}\}, Declares, E_{\mathrm{P2}}, \{(\mathtt{A}, \mathtt{b}), (\mathtt{A}, \mathtt{c})\})$

If we apply the range operator to the projection of this fragment, we see that it correctly produces all the members of class A:

$$\mathrm{ran}(\{(\mathtt{A}, \mathtt{b}), (\mathtt{A}, \mathtt{c})\}) = \{\mathtt{b}, \mathtt{c}\}.$$

We can also use the universal range to capture all the callers of `f`.

Callers of f
$(\{\mathtt{f}\}, CalledBy, E_{\mathrm{P2}}, \{(\mathtt{f}, \mathtt{c}), (\mathtt{f}, \mathtt{e})\})$

Even though relation *CalledBy* is simply the transpose of relation *Calls*, there is additional value in specifying $\mathit{CalledBy}$ as part of a model, because it allows the use of the meaning of the relation to describe fragments. Without the *CalledBy* relation, it would be difficult to intuitively represent the intent to capture all of the calls to method `f` in a single fragment.

Finally, it is possible to specify even more extensive fragments through the specification of a domain through a fragment. For example, to capture all calls by methods of class A, we can specify

Calls by methods of A
$(\mathrm{ran}(\mathrm{proj}((\{\mathtt{A}\}, \mathit{Declares}, E_{\mathrm{P2}}, P2_{J2}))), \mathit{Calls}, E_{\mathrm{P2}}, \{(\mathtt{c}, \mathtt{b}), (\mathtt{c}, \mathtt{c}), (\mathtt{c}, \mathtt{f})\})$

Obtaining the range of this last fragment's projection, we get:

$$\mathrm{ran}(\{(\mathtt{c}, \mathtt{b}), (\mathtt{c}, \mathtt{c}), (\mathtt{c}, \mathtt{f})\}) = \{\mathtt{b}, \mathtt{c}, \mathtt{f}\}.$$

The last operation we define on fragments is the *participants* operation. For any fragment, it produces a set of elements involved in the fragment.

Definition 7 (Participants) *Let* $f_P = (\mathit{Dom}, n, \mathit{Ran}, \mathit{Proj})$ *be a fragment.*

$$\mathrm{participants}(f_P) = \mathrm{dom}(\mathit{Proj}) \cup \mathrm{ran}(\mathit{Proj})$$

2.2.3 Concerns

With fragments, it is possible to express different interactions between program elements. By accumulating fragments, we can capture an increasingly large subset of a program model. However, a flat structure consisting of a list of fragments does not allow us to capture different and potentially related concerns. Often, when investigating source code, developers must reason about the code implementing concerns that are related because they are involved in a same task; other times, developers must consider the code implementing concerns that are related through a specialization relationship, where one concern addresses a specific subset of a more general concern. It is thus desirable to define a means of organizing fragments.

To address this requirement, our model includes a way to classify fragments into potentially overlapping sets. To do this, we define the notion of *concern representation* (or simply, concern) recursively, as a set of fragments and a set of *subconcerns*.

Definition 8 (Concern) *Let* P_M *be a program model. A concern* $C_P = (F_P, S_P)$ *defined on* P_M *is a tuple comprising a set of fragments* $F_P = \{f_1, f_2, ..., f_n\}$ *and a set of concerns defined on* P_M*,* $S_P = \{s_1, s_2, ..., s_m\}$.

The only constraint on the composition of fragments into a concern representation is that all of the fragments be defined on the same program model P_M. We then say that a concern is defined on P_M. Either or both of F or S can be the empty set. A fragment in F can also be in any subconcern $s \in S$. Fragments and concerns are composed into other concerns based on the requirements of a user of the representation. A root concern, not included in any parent concern, represents the broadest abstraction for a particular concern. It is called a *concern graph*.

The participants of a concern are defined as any element participating in a fragment within the concern.

Definition 9 (Concern Participants) *Let $C = (F, S)$ be a concern, where $F = \{f_1, f_2, ..., f_n\}$ is a set of fragments and $S = \{s_1, s_2, ..., s_m\}$ a set of concerns. The participants of C are defined as:*

$$\text{participants}(C) = \bigcup_{i=1}^{n} \text{participants}(f_i) \bigcup_{j=1}^{m} \text{participants}(s_j)$$

As an example of the organization of fragments into concerns, let us return to the example of program P2 (Figure 2.6). Say we are interested in investigating the uses of classes `A` and `D`. We first define a concern graph G based on the model P2_{J2} (Figure 2.7). Then we define two subconcerns, **Uses of A** and **Uses of D**. We thus have $G = (\emptyset, \{$**Uses of A**, **Uses of D**$\})$, where both subconcerns are currently empty. To complete the concern graph description we add fragments describing all calls to methods of class `A` to **Uses of A**, and all calls to methods of class `D` to **Uses of D**, respectively. We now have:

Uses of A $= ((\text{ran}(\text{proj}((\{\texttt{A}\}, \textit{Declares}, E_{\text{P2}}, P2_{J2}))), \textit{CalledBy}, E_{\text{P2}}, \{(\texttt{b}, \texttt{c}), (\texttt{c}, \texttt{c})\}), \emptyset)$
Uses of D $= ((\text{ran}(\text{proj}((\{\texttt{D}\}, \textit{Declares}, E_{\text{P2}}, P2_{J2}))), \textit{CalledBy}, E_{\text{P2}}, \{(\texttt{f}, \texttt{c}), (\texttt{f}, \texttt{e})\}), \emptyset)$

The participants of subconcern **Uses of A** are thus the methods `b` and `c`, and the participants of subconcern **Uses of D** are the methods `c,e` and `f`. As expected, the set of participants for concern graph G is the union of both sets: $\{\texttt{b}, \texttt{c}, \texttt{e}, \texttt{f}\}$.

2.3 Analyses

This section describes different operations and analyses that can be performed on concern representations.

2.3.1 Concern Analysis

Given the flexibility afforded in the composition of fragments into concerns, two concern representations can potentially overlap or be related. Given two concern representations defined on a common program model, we define their common participants as any program element participating in both concerns.

Definition 10 (Common Participants) *Let C_P and D_P be two concerns defined on a program model P_M, the set of common participants is defined as:*

$$\text{common}(C_P, D_P) = \text{participants}(C_P) \cap \text{participants}(D_P)$$

Even if two concerns have no element in common, they can still interact. We define the interaction between two concerns, defined on a common program model P_M, as the set of all modeled relations between an element in one concern and an element in the other concern.

Definition 11 (Concern Interaction) *Let C_P and D_P be two concerns defined on a program model $P_M = (E_P, N_P)$. The interaction between C_P and D_P is defined as:*

$$\begin{aligned}\text{interaction}(C_P, D_P) \quad = \quad & \{(x, n, y, \{(x, y)\}) \mid x \in \text{participants}(C_P) \wedge \\ & y \in \text{participants}(D_P) \wedge \\ & \exists\, (n, R) \in N_P : (x, y) \in R\}\end{aligned}$$

In other words, the interaction between two concerns is a set of primitive fragments representing the relations between the participants of one concern and the participants of the other concern.

The interactions between participants can also be defined for a single concern, enabling us to establish a closure of interactions between the participants of a concern. Specifically, given a concern C, the operation $\text{interaction}(C, C)$ produces a set of primitive fragments representing all the interactions between participants of C.

2.3.2 Inconsistency Management

Since concern graphs are defined on a specific program model, any change to the program impacting the model may render a concern graph inconsistent with the new program model corresponding to the changed source code. Such inconsistencies can be formally defined through a boolean function $\text{IsInconsistent}(x, P_M)$ where P_M is a program model and x a set of elements, a fragment, or a concern.

Definition 12 (Element set Inconsistency) *Let $P1_M = (E_{P1}, N_{P1})$ and $P2_M = (E_{P2}, N_{P2})$ be the models corresponding to two versions of a program produced with the same mapping function M. Let $x \subseteq E_{P1}$.*

$$\text{IsInconsistent}(x, P2_M) = x \not\subseteq E_{P2}$$

Definition 13 (Fragment Inconsistency) *Let $P1_M = (E_{P1}, N_{P1})$ and $P2_M = (E_{P2}, N_{P2})$ be the models corresponding to two versions of a program produced with the same mapping function M. Let $f_{P1} = (Dom_{P1}, n_{P1}, Ran_{P1}, Proj_{P1})$ be a fragment defined on $P1$.*

$$\begin{aligned}\text{IsInconsistent}(f_{P1}, P2_M) \quad = \quad & \text{IsInconsistent}(Dom_{P1}, P2_M) \vee \\ & \text{IsInconsistent}(Ran_{P1}, P2_M) \vee \\ & \text{Proj}_{P1} \neq \text{proj}(Dom_{P1}, n_{P1}, Ran_{P1}, P2_M).\end{aligned}$$

In other words, a fragment is inconsistent with a program model if either of its domain or range is inconsistent, or if its projection does not match the equivalent projection on the new program model. This support for detection of inconsistencies is the main justification for the existence of projections. Fragment projections store only the minimal subset of a program model required to check for inconsistencies with a different model.

Given the above definitions, we can define the inconsistency operator for concerns.

Definition 14 (Concern Inconsistency) *Let $P1_M = (E_{P1}, N_{P1})$ and $P2_M = (E_{P2}, N_{P2})$ be the models corresponding to two versions of a program produced with the same mapping function M. Let $C_{P1} = (F_{P1}, S_{P1})$ be a concern defined on $P1_M$.*

$$\begin{aligned}\text{IsInconsistent}(C_{P1}, P2_M) \quad = \quad & \exists f \in F_{P1} \mid \text{IsInconsistent}(f, P2_M) \vee \\ & \exists s \in S_{P1} \mid \text{IsInconsistent}(s, P2_M).\end{aligned}$$

Finally, it is possible to define, at the level of the concern graph model, the conditions in which an inconsistency between a fragment and a model can be automatically repaired, and the semantics of the repair operation. This way, we can ensure a common behavior for inconsistency repair across programming languages and tools supporting the concern graph model. A repairable fragment is defined as a fragment for which both the domain and the range are consistent (i.e., the fragment is only inconsistent in terms of its projection in the new program model). The repair operation is modeled as a function taking as parameters a repairable program fragment defined on a model and inconsistent with a second model, and the second model. The operation returns a fragment with the same intent as the original, but that is consistent with the second program model.

Definition 15 (Fragment Repair Operator) *Let $P1_M = (E_{P1}, N_{P1})$ and $P2_M = (E_{P2}, N_{P2})$ be the models corresponding to two versions of a program produced with the same mapping function M. Let $f_{P1} = (Dom_{P1}, n_{P1}, Ran_{P1}, Proj_{P1})$ be a fragment defined on $P1_M$ such that:*

$$\begin{aligned}&\text{IsInconsistent}(f_{P1}, P2_M) \wedge \\ &\neg\text{IsInconsistent}(Dom_{P1}, P2_M) \wedge \\ &\neg\text{IsInconsistent}(Ran_{P1}, P2_M)\end{aligned}$$

We have

$$\text{repair}(f_{P1}, P2_M) = (Dom_{P1}, n_{P1}, Ran_{P1}, \text{proj}(Dom_{P1}, n_{P1}, Ran_{P1}, P2_M)).$$

In informal terms, the repair function simply replaces the inconsistent projection of a fragment with a new projection consistent with the second program model. The practical implications of the inconsistency management support intrinsic to concern graphs are described in detail in Section 3.2.2.

2.4 Summary

In the light of the complete definition of our concern graph model, we now briefly revisit and discuss the design goals presented in Section 2.1.

Language independence In our model, concern graphs are defined on a program model that abstracts the details of specific programming languages. Concern graphs can thus be defined for programs in any language that can be modeled as a set of elements and a set of relations on these elements. Although a complete survey of the applicability of our model to different programming languages is outside the scope of this dissertation, we expect that most imperative languages, and possibly many others, can meet this simple criterion.

Flexibility All the features of our model (e.g., fragment definition, interaction analysis, inconsistency detection) are based solely on the basic definition of a program model $P = (E, N)$. It is thus possible, through the definition of a mapping function, to include arbitrarily complex relations between elements as part of the model. The level of information that can be recorded by a concern graph is thus under the control of users of the model.

Precision The goal of precision implied that there should exist a non-ambiguous mapping between any structure present in a concern representation and the corresponding source code. In our model, the relations in a program model are obtained through analysis functions $a(n, P)$ defined by the mapping function used to instantiate the model for a language. Assuming the availability of such functions implies the existence of a corresponding function capable of mapping a relation back to the corresponding source code.

Simplicity The mechanism by which we compose fragments into a concern graph is limited to the simple inclusion operation. The use of logic operators, such as the negation operator, is not supported by the model. Given a concern and a fragment, the only reasoning required from developers is to determine whether the fragment should be included or excluded from the concern.

Robustness As in the case of language independence, the goal of robustness is achieved by the use of an abstract program model to describe concerns. Because fragments record relations between program elements, as opposed to concrete references to a source code artifact (e.g., lines of code), minor changes to a program, such as re-ordering function definitions in a file, leaves the program model unchanged, and as such does not impact descriptions based on this model.

Tolerance to Inconsistencies Tolerance to inconsistencies is explicitly supported by our model. Section 2.3.2 describes the mechanism by which we can detect and repair inconsistencies between a concern graph and a program model.

Chapter 3

Tool Support for Concern Graphs

In the previous chapter we presented a general model for capturing descriptions of scattered concern code as artifacts called concern graphs. To use concern graphs effectively developers must be able to interactively specify, view, analyze, and manage concern representations for large programs. Providing support for these tasks requires the extraction of a model from a concrete program. In turn, the automatic extraction of a program model requires a definition of a mapping function that can produce models that are both useful, usable, and scalable.

To experiment with concern graphs, we developed support for using concern graphs with Java programs. Based on a combination of experience, experimentation, and the work of other researchers, we designed a mapping function that produces models which allow developers to describe a variety of concerns in source code. In Section 3.1 we present the mapping function we have designed and discuss and justify our choices in elaborating this mapping function. Then, in Section 3.2, we describe a tool we have developed to support concern graphs according to our mapping function for Java.

3.1 General Mapping Function for Java

A mapping function specifies how to produce a program model on which concern graphs can be specified (see Chapter 2). Many different mapping functions can be defined for a programming language, each one presenting a tradeoff between, on one hand, the expressiveness of a model to represent details of a program, and on the other, simplicity, usability, and scalability. The mapping function we have designed to support concern graphs for Java is intended to be both scalable and capable of representing a wide range of concerns. This mapping function, named Java Standard, is presented in Figure 3.1. The detailed definition of the boolean functions involved in the mapping function are presented in Appendix B.

The program elements captured by the Java Standard mapping function are limited to classes, interfaces, fields, and methods. Local (intra-method) elements, such as method parameters and local variables, are not captured in models produced by the mapping function. We decided not to consider intra-method elements for two main reasons. First, we wanted to establish a practical bound to the size and complexity of models required to define concern graphs, so that the approach would remain usable and scalable. Second, intra-method program elements are not considered because we are mostly interested in capturing scattered concerns, that is, concerns presenting interactions not limited to a module.

Mapping Function Java Standard

$$
\begin{aligned}
E &= \{x \mid \text{IsAClass}(x) \wedge \text{IsAnInterface} \wedge \text{IsAField}(x) \wedge \\
&\quad \text{IsAMethod}(x)\} \\
\text{names}(N) &= \{\mathit{Accesses}, \mathit{AccessedBy}, \mathit{Calls}, \mathit{CalledBy}, \mathit{Checks}, \\
&\quad \mathit{Creates}, \mathit{Declares}, \mathit{ExtendsClass}, \mathit{ClassExtendedBy}, \\
&\quad \mathit{ExtendsInterface}, \mathit{InterfaceExtendedBy}, \\
&\quad \mathit{HasParameterType}, \mathit{HasReturnType}, \mathcal{I}, \mathit{Implements}, \\
&\quad \mathit{ImplementedBy}, \mathit{OfType}, \mathit{Overrides}, \mathit{OverridenBy}, \\
&\quad \mathit{TransitivelyExtends}, \mathit{TransitivelyExtendedBy}, \\
&\quad \mathit{TransitivelyImplements}, \mathit{TransitivelyImplementedBy}\} \\
a(\mathit{Accesses}, P) &= \{(x, y) \mid \text{Accesses}(x, y)\} \\
a(\mathit{AccessedBy}, P) &= a(\mathit{Accesses}, P)^{\top} \\
a(\mathit{Calls}, P) &= \{(x, y) \mid \text{Calls}(x, y)\} \\
a(\mathit{CalledBy}, P) &= a(\mathit{Calls}, P)^{\top} \\
a(\mathit{Checks}, P) &= \{(x, y) \mid \text{Checks}(x, y)\} \\
a(\mathit{Creates}, P) &= \{(x, y) \mid \text{Creates}(x, y)\} \\
a(\mathit{Declares}, P) &= \{(x, y) \mid \text{Declares}(x, y)\} \\
a(\mathit{ExtendsClass}, P) &= \{(x, y) \mid \text{ExtendsClass}(x, y)\} \\
a(\mathit{ClassExtendedBy}, P) &= a(\mathit{ExtendsClass}, P)^{\top} \\
a(\mathit{ExtendsInterface}, P) &= \{(x, y) \mid \text{ExtendsInterface}(x, y)\} \\
a(\mathit{InterfaceExtendedBy}, P) &= a(\mathit{ExtendsInterfaces}, P)^{\top} \\
a(\mathit{HasParameterType}, P) &= \{(x, y) \mid \text{HasParamterType}(x, y)\} \\
a(\mathit{HasReturnType}, P) &= \{(x, y) \mid \text{HasReturnType}(x, y)\} \\
a(\mathcal{I}, P) &= \{(x, y) \mid x = y\} \\
a(\mathit{Implements}, P) &= \{(x, y) \mid \text{Implements}(x, y)\} \\
a(\mathit{ImplementedBy}, P) &= a(\mathit{Implements}, P)^{\top} \\
a(\mathit{OfType}, P) &= \{(x, y) \mid \text{OfType}(x, y)\} \\
a(\mathit{Overrides}, P) &= \{(x, y) \mid \text{Overrides}(x, y)\} \\
a(\mathit{OverridenBy}, P) &= a(\mathit{Overrides}, P)^{\top} \\
a(\mathit{TransitivelyExtends}, P) &= a(\mathit{ExtendsClass}, P)^{+} \\
a(\mathit{TransitivelyExtendedBy}, P) &= a(\mathit{TransitivelyExtends}, P)^{\top} \\
a(\mathit{TransitivelyImplements}, P) &= (a(\mathit{ExtendsClass}, P)^{*} \circ a(\mathit{Implements}, P)) \cup \\
&\quad (a(\mathit{ExtendsClass}, P)^{*} \circ a(\mathit{Implements}, P) \circ \\
&\quad a(\mathit{ExtendsInterface}, P)^{+})
\end{aligned}
$$

Figure 3.1: The mapping function Java Standard

The first reason, to limit the model size, comes from the realization that the more expressive a program model is, the higher the computational and memory cost to produce it, and the higher the human effort required to use it. From the perspective of memory cost, detailed program models inevitably comprise more nodes and edges than more abstract ones. The inclusion of local variables and detailed control- and data-flow relations into a program model, such as the program dependence graph [40], can seriously impact the scalability of the representation. For example, in their work on chopping (a variant of slicing), Jackson and Rollins noted this problem with the dependence graphs required to manage intra-module relations: "Graphs of even the smallest chops tend to be huge" [62: p. 9]. Producing precise and fine-grained program models also incurs a non-significant cost in terms of computational time [118]. Finally, with an increase in the variety of program element types supported by a model comes a significant increase in the number of potential relation types between elements that a developer must consider (e.g., data-dependence, control-domination).

The second argument for not considering intra-method elements is that in most cases they are not needed to model scattered concerns, and the cost of their inclusion in the model is, as such, unwarranted. Specifically, since elements such as local variables cannot be referenced by elements outside the method, they are not useful for describing a concern scattered in multiple methods. As the designers of the C Information Abstractor tool have noted, “Details of interactions between local objects are ignored because they are only interesting in a small context” [25: p. 326]. We followed a similar philosophy in elaborating the design of our mapping function.

We categorize the 22 relations supported by the mapping function as either *structural* or *behavioral*. Structural relations represent static, declarative relations between elements in a program. Roughly speaking, static relations are the type of relations that would be documented in a UML static structure diagram [51]. As explained in Section 2.2.2, the identity relation ($\mathcal{I}$) is an artificial relation used to include individual elements in a concern graph. The *Declares* relation expresses the basic declarative structure of a program. This relation can be used to specify fragments representing classes or interfaces with all their members. Additional relations exposing the declarative structure of a program are the *HasParameterType*, *HasReturnType*, and *OfType* relations. These relations expose, respectively, the parameter types of a method, the return type of a method, and the type of a field. The transpose of relations exposing the declarative structure of a program are not included in the mapping function because we could not foresee any use of these relations. The relations *ExtendsClass*, *ExtendsInterface*, and *Implements* (and their respective transpose) expose the basic class hierarchy of a program. Because of the coarse granularity of such relations, we included a transitive version of the *Extends* and *Implements* relations. This way, it is possible to specify an inheritance or implementation relation between two classes (and/or interfaces) even if the classes are not in direct relation with each other. Finally, the *Overrides* relation and its transpose expose whether a method is substitutable for another one at run-time.

Behavioral relations represent code within a method. The *Accesses* relation and its transpose represent code reading or writing to a field. The *Calls* relation and its transpose represent method calls. The *Creates* relation represents the creation of a new object using the keyword `new`, and the *Checks* relation represents a downcast or the comparison of the run-time type of an object with a certain type. The transpose of the *Checks* and *Creates* relations are not included in the mapping function because it is not clear how useful fragments defined using these relations would be in specifying concerns. Using the behavioral relations above, it is possible to specify a subset of a method as part of a concern. For example, Figure 3.2 shows the code of a method that resets the state of an object. To capture all of the code dealing with accessing object state, we can use the *Calls* relation, such as in the fragment

```
resetState Calls getFlag(int).
```
[1]

[1]For the examples in this section, we use a simplified representation for fragments, and we omit the fully-qualified names of Java elements that would normally be present. For the complete description of fragments structures, see Section 2.2.2.

This fragment captures the method calls to `getFlag()` on lines 3–6. To capture the code dealing with the state of the `AUTOSAVE_DIRTY` flag, we can use the *Accesses* relation, such as in the fragment

`resetState() Accesses AUTOSAVE_DIRTY`.

This fragment captures code on lines 3 and 9.

```
1:  public void resetState()
2:  {
3:     if( !getFlag(AUTOSAVE_DIRTY )
4:        || !getFlag(DIRTY)
5:        || getFlag(LOADING)
6:        || getFlag(IO))
7:        return;
8:
9:     setFlag(AUTOSAVE_DIRTY,false);
10: }
```

Figure 3.2: Method `OptionGroup.save()`

One of the characteristics of models produced with the Java Standard mapping function is that they do not support the distinction between different contexts in source code corresponding to a behavioral relation. For example, in the method of Figure 3.2, it is not possible to include the call to `getFlag(int)` on line 6 as part of a concern, while excluding the other calls to `getFlag(int)`. Context sensitivity of this form would require a more detailed program model [140]. As we will explain in Chapter 4, context-insensitivity has been a reasonable choice because when a call to a non-library method contributes to the implementation of a concern, most of the calls to that method are usually part of the concern as well. In situations where this has not been the case, the small number of false positives have not caused problems with the task.

Finally, the mapping function does not support exception handling. Exception handling introduces a particular type of control-flow that can be difficult to abstract [27, 111, 114, 125]. For the purpose of experimenting with concern graphs and validating the thesis, we chose to leave exception handling aside. Although this prevents users of the approach to specify concerns related to error handling, there exists many other possible types of concerns.

3.2 The Feature Analysis and Exploration Tool

To support the task of finding the source code implementing concerns of interest to a developer, and of representing those concerns with concern graphs, we built the Feature Exploration and Analysis Tool (FEAT). FEAT supports three main functions.

1. **Model Extraction** It extracts a model of a program based on the mapping function Java Standard described in Section 3.1, and provides a user of the tool access to the model.

2. **Concern Construction** It allows a user to build and modify concern representations by specifying fragments on the model extracted from a program. It supports the saving of a concern representation to permanent storage, and the loading of a concern representation in the tool.

3. **Analysis** It supports the analysis of the interactions between different concerns. It also supports the detection and repair of inconsistencies between a concern graph and a program.

To integrate building, viewing, and modifying concern graphs with the activities of code investigation and modification, we have implemented the FEAT tool as a plug-in for the Eclipse Platform [93]. Eclipse is an integrated development environment for Java with an architecture that supports the addition of modules (called plug-ins) that add to the development environment's functionality. With the FEAT plug-in installed in Eclipse, developers can use the integrated development environment as usual, to browse and modify source code, perform searches, etc. However, if a user desires to create a concern representation, the functionality provided by the FEAT plug-in is activated, providing the three functions described above.

3.2.1 Usage Model

When investigating and modifying source code during a program evolution task, a developer typically starts using the features of the FEAT plug-in when a concern of interest is identified. In general, for a developer working on a non-familiar code base, the use of the FEAT tool proceeds as follows:

1. **Broad investigation outside FEAT** A developer performs broad searches in an attempt to discover an area of the code related to the modification task. For example, a developer asked to implement an enhancement to the autosave feature of the jEdit application described in Section 1.2 might perform a lexical search for the keyword "autosave" on all the source code files. This type of general investigation does not focus on any particular concern and is usually performed using the basic features of the Eclipse platform, and without help from the FEAT plug-in.

2. **Identification of a concern** When trying to understand the code related to a modification task, a developer realizes that the code related to the modification implements one or more concerns that need to be considered [8]. For example, while preparing for the autosave enhancement task, a developer might come across the PROPERTIES MANAGEMENT concern. At this point, the FEAT tool can be used to capture the implementation of the concern.

3. **Creating a concern graph** Using a menu in the user interface, a developer creates a concern graph associated with a code base (also called a *project* in Eclipse). Creating a concern graph initializes the FEAT tool and extracts the program model for the code base.

4. **Seeding the concern graph** When a new concern graph is created, it is originally empty (i.e., it describes no source code). Elements (classes, methods, or fields) from the code base must be moved to the concern graph. This process is called *seeding* the concern graph. For example in the case of the autosave task, a method relevant to MANAGING PROPERTIES, `jEdit.setProperty(String,String)` can be added to the concern.

5. **Building the concern graph** Once a concern graph is seeded, queries are performed on the elements in the concern graph, to elicit the relations between elements in the concern and the rest of the code base. For example, a FEAT query can reveal all the callers of the `jEdit.setProperty(String,String)` method. Query results that are relevant to the concern are added to the concern representation in FEAT.

6. **Analyzing the concern graph** If necessary, a concern graph can be divided into different sub-concerns. While building a concern graph, a developer can then add elements and relations specifically to the sub-concern under investigation. Returning to the example of the autosave task, the concern graph for the task can be subdivided into a SAVING WIDGET STATE concern and a PROPERTIES MANAGEMENT concern. When a concern comprises different sub-concerns, it is possible to analyze the interactions between elements in the different sub-concerns.

7. **Saving the concern graph** When the concern graph captures enough of the implementation of the concerns of interest, it can be saved to disk.

The steps above represent a simplified process. In practice, many variations can take place. For example, instead of creating a new concern graph, a developer can load an existing concern graph produced as part of a prior task. Additionally, investigation and concern graph construction activities within the FEAT tool can be interleaved with basic Eclipse searches and code modifications. The next section describes in more detail how users interact with the FEAT tool.

3.2.2 User Interface

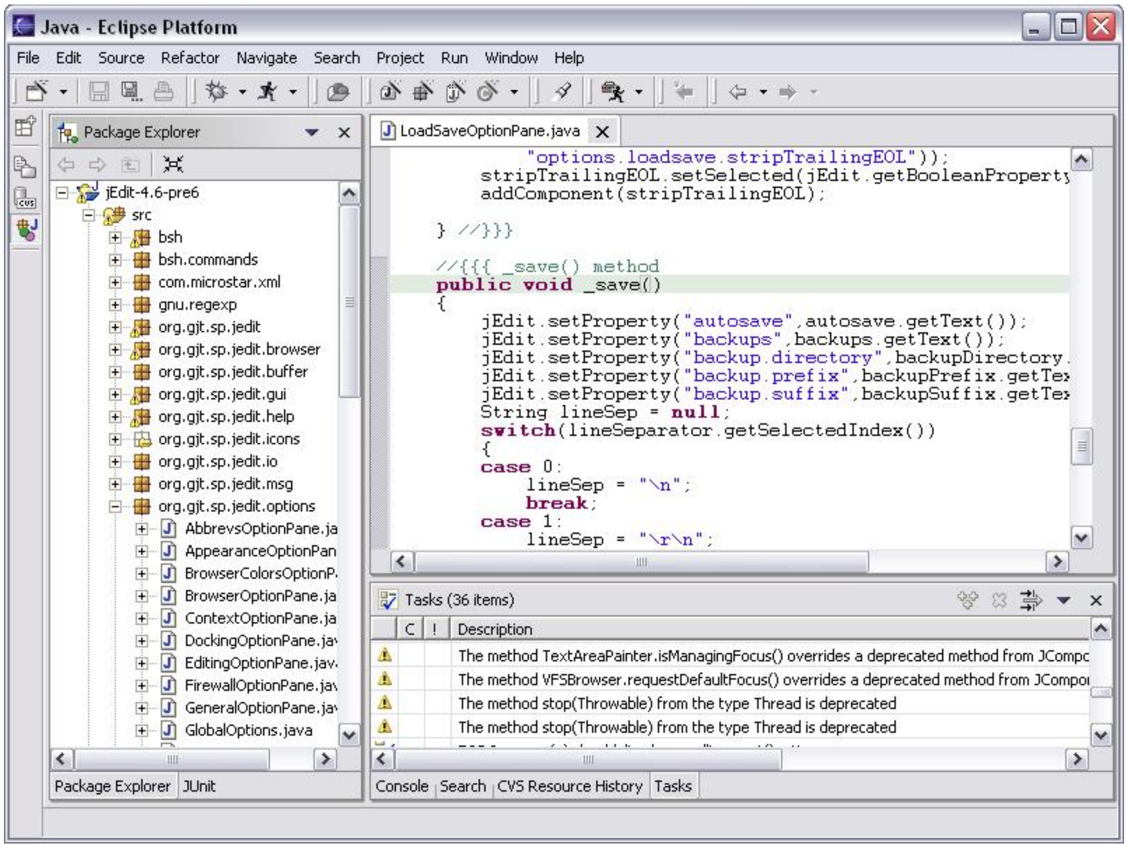
The description of the user interface of the FEAT plug-in (version 2.3.0) focuses on how a user interacts with the FEAT tool when performing four principal tasks: viewing a concern graph, exploring the code and building a concern graph, comparing concerns, and managing the inconsistencies between a concern graph and the source code.[2] Although the four tasks are separated for the purpose of their description, they would, in practice, be overlapping. To set the context for the tool, we first describe the Eclipse platform.

Eclipse

In Eclipse, functionality is provided at two different levels: the *workbench* level, and the *view* level. The workbench is the main application window (Figure 3.3). The workbench is the interface to a collection of *resources*, called the *workspace*. Resources in the workspace correspond to files or directories on a system. For example, Java source code files are typical Eclipse resources. Within the workspace, resources are organized into different *projects*. The workbench is the user interface that provides general-purpose functionality, such as opening and closing resources, performing searches, etc. Within the workbench, more specialized functionality is provided through different *views*. A view is a user interface window that displays some data and that provides operations on this data. For example, in Figure 3.3, the window on the left is a view called the Package Explorer. It presents a hierarchical view of the different packages in a Java project, of the Java source code files in each package, and of the elements (classes and class members) declared in each file. The Package Explorer also supports operations on the elements visible in the view, such as cross-reference searches on an element. Each view has a separate tool bar that provides operations specific to the data in the view. For example, the tool bar for the Package Explorer allows a user to filter out certain types of elements from the view. *Editors* are a special type of view that allow users to modify resources. A

[2]Readers interested in the details of the FEAT user interface can consult the manual distributed with the FEAT tool [109].

collection of views addressing a specific purpose is called a *Perspective*. In Figure 3.3, the active perspective is the Java perspective. The Java perspective includes views supporting Java development, and an editor area. Users can switch between different perspectives using the vertical tool bar on the left of the workbench window. Switching perspectives does not affect the state of the resources in the workspace.



```
            "options.loadsave.stripTrailingEOL"));
        stripTrailingEOL.setSelected(jEdit.getBooleanPropert
        addComponent(stripTrailingEOL);

    } //}}}

    //{{{ _save() method
    public void _save()
    {
        jEdit.setProperty("autosave",autosave.getText());
        jEdit.setProperty("backups",backups.getText());
        jEdit.setProperty("backup.directory",backupDirectory.
        jEdit.setProperty("backup.prefix",backupPrefix.getTe
        jEdit.setProperty("backup.suffix",backupSuffix.getTe
        String lineSep = null;
        switch(lineSeparator.getSelectedIndex())
        {
        case 0:
            lineSep = "\n";
            break;
        case 1:
            lineSep = "\r\n";
```

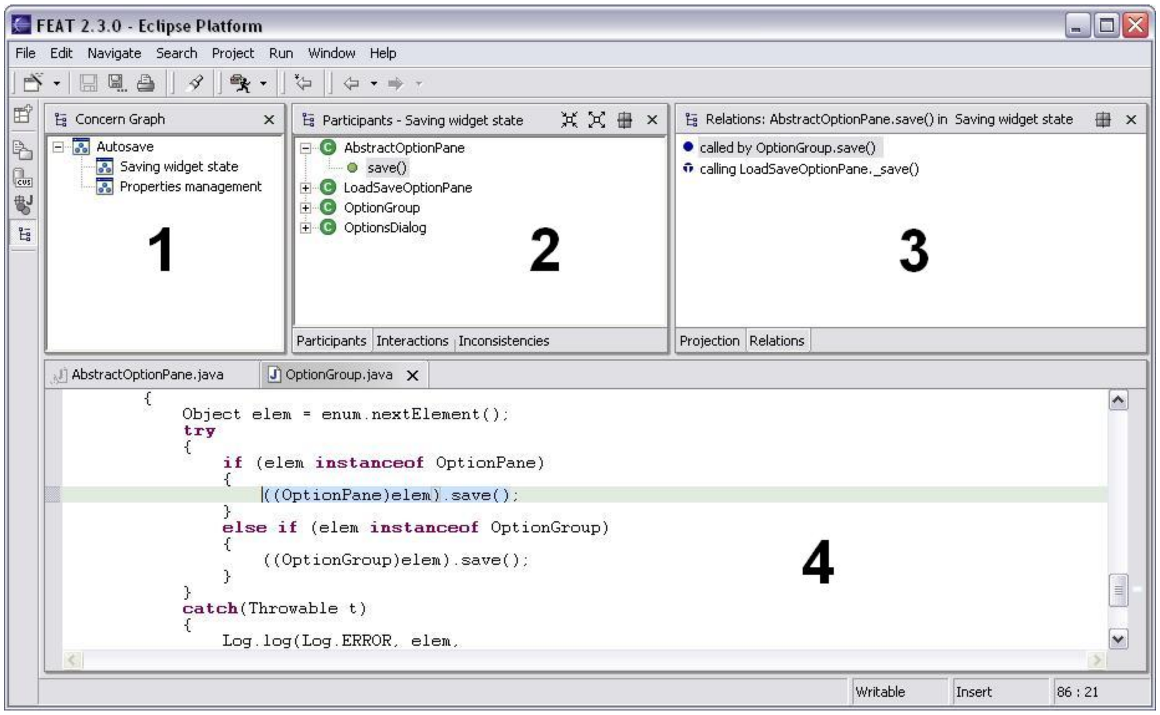
The method TextAreaPainter.isManagingFocus() overrides a deprecated method from JCompc
The method VFSBrowser.requestDefaultFocus() overrides a deprecated method from JCompo
The method stop(Throwable) from the type Thread is deprecated
The method stop(Throwable) from the type Thread is deprecated

Figure 3.3: The Eclipse platform

Viewing a Concern Graph

A user views an existing concern graph by switching to the FEAT Perspective. The FEAT Perspective is a collection of views showing a concern graph in decreasing levels of abstraction (Figure 3.4).

The Concern Graph View (area 1) shows the hierarchy of concerns for a concern graph (see Section 2.2.3). From this view, users can create new child concerns, delete existing concerns, and move concerns in the hierarchy. Figure 3.5 shows a concern hierarchy for the task of enhancing the autosave feature in the jEdit application (see Section 1.2). This hierarchy consists of a top-level concern (or concern graph) named AUTOSAVE, and two sub-concerns, SAVING WIDGET STATE and PROPERTIES MANAGEMENT. Selecting any concern in the Concern Graph View displays all of the participants for the concern (see Definition 9). The concern selected in the Concern Graph View is called the *active concern*. The participants for the active concern are displayed in the Participants View (area 2).


FEAT 2.3.0 - Eclipse Platform
File Edit Navigate Search Project Run Window Help
Concern Graph
Autosave
Saving widget state
Properties management
1
Participants - Saving widget state
AbstractOptionPane
save()
LoadSaveOptionPane
OptionGroup
OptionsDialog
2
Participants Interactions Inconsistencies
Relations: AbstractOptionPane.save() in Saving widget state
called by OptionGroup.save()
calling LoadSaveOptionPane._save()
3
Projection Relations
AbstractOptionPane.java
OptionGroup.java
4
Writable
Insert
86 : 21

```
        {
            Object elem = enum.nextElement();
            try
            {
                if (elem instanceof OptionPane)
                {
                    ((OptionPane)elem).save();
                }
                else if (elem instanceof OptionGroup)
                {
                    ((OptionGroup)elem).save();
                }
            }
            catch(Throwable t)
            {
                Log.log(Log.ERROR, elem,
```

Figure 3.4: The FEAT Perspective. Area 1 holds the Concern Graph View. Area 2 holds the Participants, Interactions, and Inconsistencies Views. Area 3 holds the Projection and Relations Views. Area 4 holds the Java Editor.

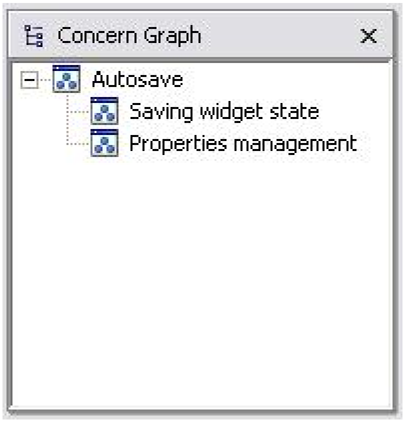


Figure 3.5: The Concern Graph View

In the Participants View, participants for a concern are displayed as a set of trees, with participant classes at the root of the trees; participant members are displayed as children of their declaring class. For example, Figure 3.6 shows the participants for the sub-concern SAVING WIDGET STATE. The participants include elements in the classes `AbstractOptionPane`, `LoadSaveOptionPane`, `OptionGroup`, and `OptionsDialog`. The nodes for the first two classes are expanded, revealing their members who participate in the concern: methods `save()` and `_save()`, respectively.

Double-clicking on any participant shows its declaration in a Java editor (area 4). Selecting a participant shows all of the relations between this participant and any other participant in the active concern (area 3). This display of the relations between participants in of a concern corresponds to the intra-concern analysis described in Section 2.3.1.

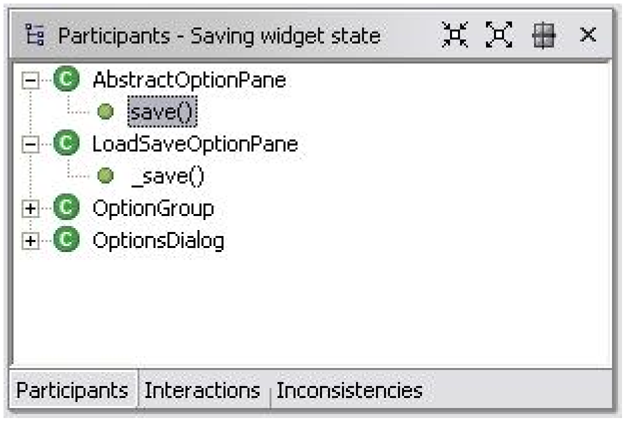


Figure 3.6: The Participants View

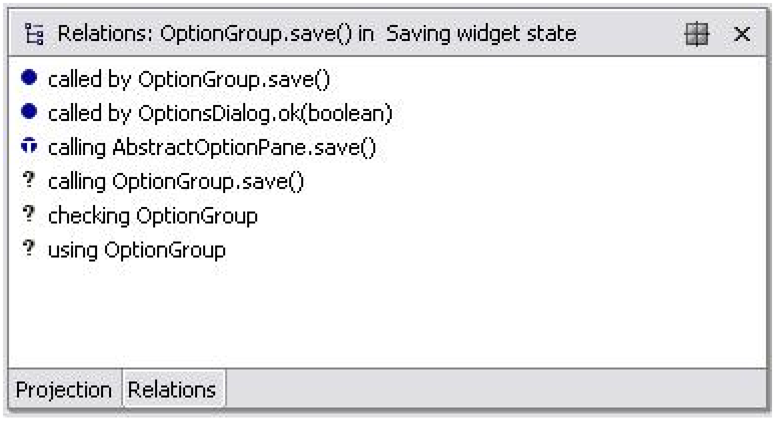
The relations for a participant selected in the Participants View are displayed in the Relations View (area 3). For example, Figure 3.7 shows the relations for participant `OptionGroup.save()`. The icon to the left of a relation indicates whether a relation is part of a fragment explicitly added by a user (as described below), or whether it was identified through intra-concern analysis. A blue dot identifies a relation explicitly added by a user. A blue dot with a white T inscribed in it identifies the transpose of a relation comprised in a fragment explicitly added to a concern. A question mark identifies a relation that was not added to a concern graph by a user, but that was discovered through intra-concern analysis. Relations identified through intra-concern analysis are displayed but are not part of a concern. However, a user can add these relations to a concern. In this case, the question mark becomes a blue dot, indicating an explicit relation. Finally, clicking on any relation shows the source code corresponding to the relation. For example, clicking on the relation `called by OptionGroup.Dialog.ok(boolean)` will bring up the code of `OptionsDialog.ok(boolean)` in the editor area and highlight the call to `OptionGroup.save()`.

Exploring the Code and Building a Concern Graph

To help a developer investigate the source code for a project, the FEAT plug-in supports a set of queries on the classes, methods, and fields declared in the project associated with a concern graph. The FEAT queries support the investigation of all of the relations specified in the mapping function Java Standard. A query in FEAT corresponds to a fragment that has a universal range (see Section 2.2.2). For example, a query to determine all of the callers of a method `m()` is modeled as the fragment

`m() CalledBy ALL.`[3]

[3]In this chapter, we use the keyword "ALL" to represent the universal range E in a program model.


Relations: OptionGroup.save() in Saving widget state

● called by OptionGroup.save()
● called by OptionsDialog.ok(boolean)
calling AbstractOptionPane.save()
? calling OptionGroup.save()
? checking OptionGroup
? using OptionGroup

Figure 3.7: The Relations View

A user performs a FEAT query by right-clicking on a Java element in any FEAT view, and choosing a relation in a pop-up menu. In the pop-up menu, queries are organized in two groups: fan-in queries and fan-out queries. The criterion for distinguishing fan-in from fan-out queries is based on the predicate “knows-about”. Fan-in queries return elements that *know about* the queried element. Fan-out queries return elements that the queried element *knows about*. For example, fan-out queries for a method include the relation *Calls*, while fan-in queries include its transpose, the relation *CalledBy*. Figure 3.8 shows a query about to be performed on element `LoadSaveOptionPane.save()` in the Participants View. The figure shows that a pop-up menu has been invoked on the element, and the menu item `calling` has been selected from the menu group `Fan-out`. This query corresponds to the fragment

```
LoadSaveOptionPane.save() Calls ALL.
```

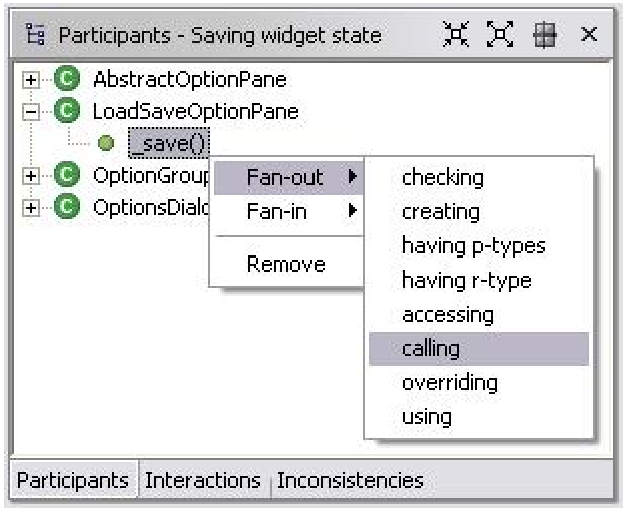


Figure 3.8: A FEAT query

Internally, FEAT queries are built and managed as fragments: performing a query consists of applying the projection operator defined in Section 2.2.2 on the database stored by the plug-in. The results of a query correspond to the projection of the fragment that represents the query. Query results are displayed in the Projection View. The Projection View is the main view used to investigate the code in FEAT. It is shown in the same area (area 3) as the Relations View in the FEAT Perspective: selecting a tab at the bottom of the area allows a user to switch between views. Figure 3.9 shows the results of the query of Figure 3.8 as presented in the Projection View. Query results are displayed in a tree representing the projection of a fragment. The elements in the tree above the relation node represent the domain of the projection. The elements below the relation node represent the range of the projection. In our example the `_save()` method calls six methods in four different classes. Double-clicking on any element in the Projection View displays its declaration in an editor. Selecting an element displays the source code for only the relation. For example, selecting method `getSelectedItem()` will display the line in the `_save()` method where `getSelectedItem()` is called. From within the Projection View, it is also possible to add elements and relations to a concern graph. To add a single element to the active concern, a user can right-click on any element in the view and select `Add element to concern`. This action will result in the addition of a single element to the active concern; the element is expressed as a primitive fragment using the identity relation. To add a query result and the corresponding relation to the active concern, a user can select any range element (i.e., below the relation node in the tree), right-click and select `Add relation to concern`. This action will add to the active concern a primitive fragment consisting of the element queried (as the domain), the relation queried, and the element selected in the Projection View (as the range). For example, in the case of Figure 3.9, if a user right-clicks on `getSelectedItem()` and selects `Add relation to concern`, the fragment

```
LoadSaveOptionPane._save() Calls JComboBox.getSelectedItem()
```

will be added to the active concern. Whenever a fragment is added to the active concern, the Participants View is updated to show the new participants. In some case, the entire results of a query will be relevant to a concern. In this case, it is possible to add the entire query result to the active concern though a menu in the tool bar of the Projection View. In this case, the fragment that is added to the active concern consists of the element queried (as the domain), the relation queried, the universal range, and the projection corresponding to the query. Finally, the Projection View preserves all of the queries performed in a history list. Users can recall the results of any previous query by selecting from a list at the top of the view.

Comparing Concerns

When a concern graph is subdivided into different sub-concerns, it is possible to analyze two concerns in the hierarchy to determine their interactions (see Section 2.3.1). To determine the interactions between two concerns, a developer selects the concerns in the Concern Graph View, right-clicks, and selects `Compare` from a pop-up menu. The results of the analysis appear in a view called the Interactions View, which overlaps with the Participants View. For example, a comparison of the concerns SAVING WIDGET STATE and PROPERTIES MANAGEMENT discussed above results in the Interactions View as depicted in Figure 3.10.

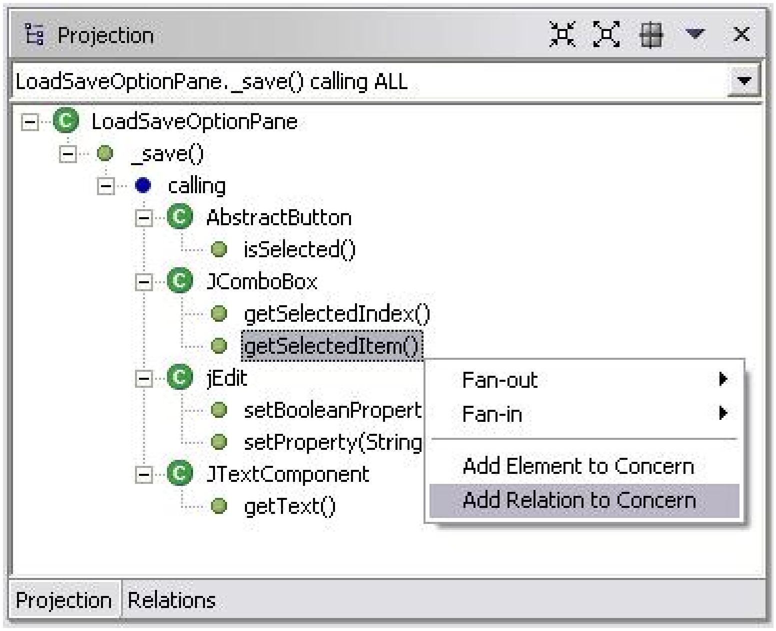


Figure 3.9: Query results in the Projection View

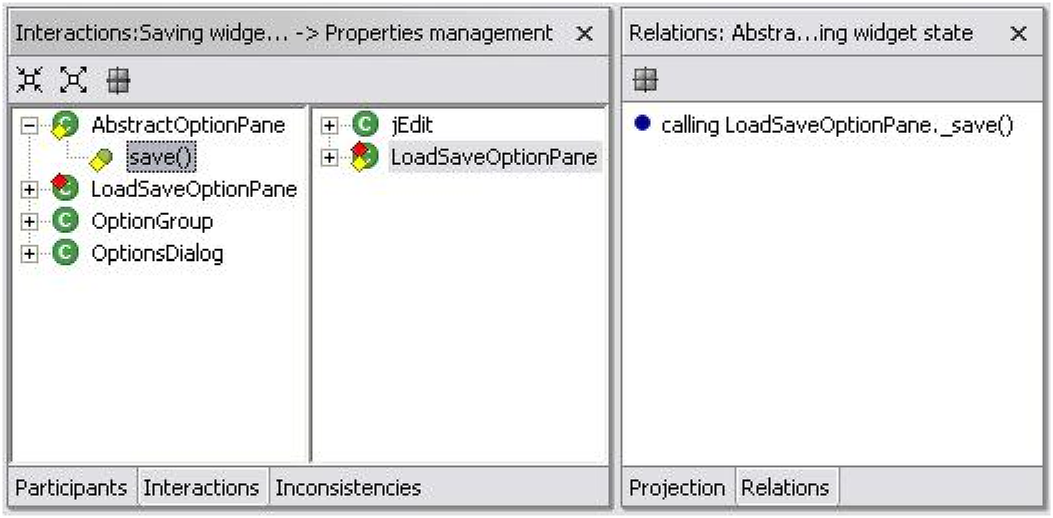


Figure 3.10: The Interactions and Relations Views

The Interactions View shows the participants of the two selected concerns side by side. Participants common to both concerns are annotated with a red diamond. Participants in one concern that are directly related to any participant in the other concern through a relation supported by the model are annotated with a yellow diamond. For example, from Figure 3.10, we can tell that class `LoadSaveOptionPane` is common to both concerns because it is annotated with a red diamond. We can also determine that method `save()` of class `AbstractOptionPane` in SAVING WIDGET STATE is related to a participant in PROPERTIES MANAGEMENT. In the Interactions View, selecting a participant shows all of the relations between the selected participant and any participant in the other concern. This display contrasts with the selection of a participant in the Participants View, which shows the relations between the participants of a single concern. For example, in Figure 3.10, selecting `AbstractOptionPane.save()` reveals that the method calls method `LoadSaveOptionPane._save()` in concern `Properties management`. Using the interaction analysis feature of FEAT, a developer can quickly focus on the areas of interactions between two concerns without having to investigate all of the concern code.

Managing Inconsistencies

The FEAT tool is tolerant of inconsistencies between a concern graph and the source code. When a concern graph is loaded into FEAT, and any time the source code changes, FEAT performs an inconsistency check for each fragment. Checking for inconsistencies consists of applying the $\mathrm{IsInconsistent}$ function defined in Section 2.3.2 (Definition 13) to each fragment. Even in the case where inconsistencies are detected, the FEAT tool functions as usual: participants in consistent concerns are displayed and can be queried and analyzed. However, the participants and relations for any inconsistent fragment are not displayed in the Participants View and the Relations View. To indicate that inconsistencies were detected, any concern containing one or more inconsistent fragment is annotated with a red icon in the Concern Graph View. Additionally, it is possible to view, query, and repair inconsistent fragments in the Inconsistency View. To display the Inconsistency View, a developer right-clicks on any inconsistent concern in the Concern Graph View and selects `Inconsistencies` from a pop-up menu. Figure 3.11 shows the Inconsistency View listing three different inconsistencies. Inconsistencies are identified by the name of the inconsistent fragment. FEAT recognizes three different types of inconsistencies:

- **Primitive Inconsistency** The relation captured by a primitive fragment does not exist in the source code. Specifically, a primitive inconsistency is detected when the $\mathrm{IsInconsistent}$ function applied to a primitive fragment returns true because any clause in Definition 13 is true. These types of inconsistencies are not automatically repairable.

- **Inconsistent Domain Inconsistency** The domain of the fragment is inconsistent. Specifically, an inconsistent domain inconsistency is detected when the $\mathrm{IsInconsistent}$ function returns true because the domain set is inconsistent (See Definition 13, Section 2.3.2). These types of inconsistencies are not automatically repairable.

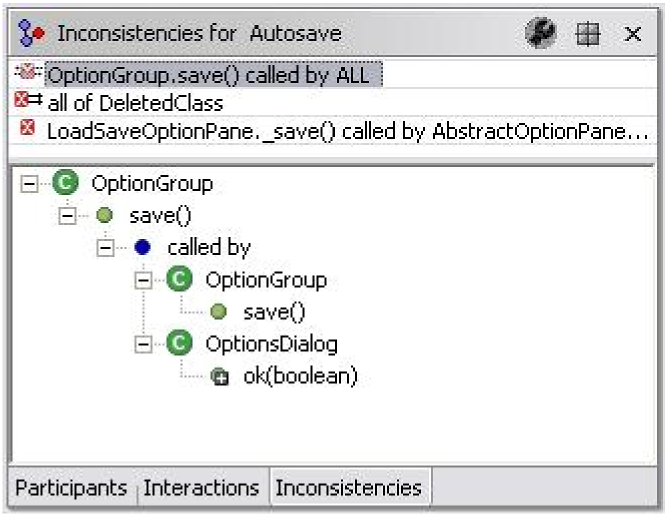
- **Projection Mismatch Inconsistency** The projection of the fragment does not match the current source code. Specifically, a projection mismatch inconsistency is detected when the $\mathrm{IsInconsistent}$ function returns true because the third clause in Definition 13 is true but the two other clauses are false. These types of inconsistencies are automatically repairable (see Definition 15, Section 2.3.2).

In the Inconsistency View, the three different types of inconsistencies are distinguished by the icon on the left of the inconsistent fragment's name. Primitive inconsistencies are identified with a red X (e.g., the third inconsistency in the list at the top of the view in Figure 3.11). Inconsistent domain inconsistencies are identified with a red X and two right arrows (e.g., second inconsistency).

Projection mismatch inconsistencies are identified with a red X superimposed on a right and a left arrow (e.g., first inconsistency). Clicking on any fragment name in the inconsistency list displays the inconsistent fragment in a tree structure in the lower part of the view. The inconsistent fragment is presented in a style similar the one used in the Projection View. Any element in the inconsistent fragment which exists in the source code can be displayed in an editor or queried as in the Participants View or the Projection View. This display allows users to investigate the relations between an element in an inconsistent fragment and the rest of the code base. As a result of such queries, a user may decide to add to the concern description based on information in the inconsistent fragment. Elements in the lower part of the Inconsistency View are annotated with an icon denoting whether the element does not exist, whether the corresponding relation exists in the code but not in the concern graph, or exists in the concern graph but not in the code. For example, Figure 3.11 displays the fragment

```
OptionGroup.save() CalledBy ALL
```

with the method `ok(boolean)` of class `OptionsDialog` annotated with a "+" icon. This icon indicates that the call from `OptionsDialog.ok(boolean)` is documented in the concern graph but does not exist in the current version of the source code.



```
OptionGroup.save() called by ALL
all of DeletedClass
LoadSaveOptionPane._save() called by AbstractOptionPane...

OptionGroup
    save()
        called by
            OptionGroup
                save()
            OptionsDialog
                ok(boolean)
```

Figure 3.11: The Inconsistency View

The Inconsistency View also allows a user to make an inconsistent concern graph consistent with the source code. Right-clicking on any fragment in the list of inconsistent fragments will bring up a pop-up menu with the item `Repair`. Repairing a repairable fragment will synchronize the fragment with the source code according to the algorithm of Section 2.3.2. For example, the fragment selected in Figure 3.11 is inconsistent due to a projection mismatch, and as such can be automatically repaired. Repairing a non-repairable fragment will remove the fragment from the concern graph. A button in the tool bar of the Inconsistency View allows a user to repair all fragments at once. This way, a concern graph can be made consistent with the source code in a single step.

3.2.3 Implementation

The architecture of the FEAT tool comprises three components: the model, the analyzer, and the user interface. Figure 3.12 illustrates the dependencies between the three components.

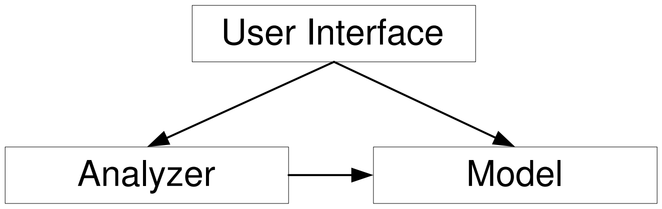


Figure 3.12: The architecture of the FEAT tool

Model

The model component supports the run-time representation of a concern graph, enabling support for loading a concern graph from permanent storage and saving a concern to permanent storage, and for tolerating inconsistencies between a concern graph and the source code. The model component is independent from the user interface or the analyzer, allowing it to be used to present concern graphs in different interfaces, and to allow third-party developers or researchers to use concerns graphs independently of the FEAT tool. The model component stores a concern graph in a structure similar to its theoretical structure. The model supports saving a concern graph to permanent storage by providing functionalities to export a concern graph to an XML document format [21]. The model also provides parsing functionality to load a concern graph from its XML representation. Finally, the model component is made tolerant to inconsistencies through a mechanism of pollution markers inspired by the work of Balzer [6]. With pollution markers, inconsistent fragments can be marked in the model. Other components that use the model can then query a fragment object to determine whether it is consistent or not, and take appropriate action.

Analyzer

The responsibility of the analyzer component is to produce a model for a program based on the mapping function Java Standard, to support queries on this model, and to support mapping primitive fragments to the corresponding source code. The analyzer component is designed to optimize the speed of FEAT queries, at the cost of an initial model extraction time.

The analyzer implemented in version 2.3.0 of the FEAT tool produces a model of a program by executing a single pass through the abstract syntax tree (AST) of every Java file in the project associated with a concern graph. The AST for Java files is provided as part of the Eclipse Platform. When scanning the AST of Java source files, the analyzer searches for instances of the relations supported by the mapping function. When a relation between two elements is identified, the analyzer stores both the relation and its transpose in an in-memory database.

To avoid performing a second analysis pass, certain relations are not stored in the model database and are instead computed on-the-fly at query time. In particular, to elicit the complete range of *Calls* relations, the analyzer must determine all of the potential bindings for a static signature at virtual method call sites. The current algorithm used to determine potential run-time method bindings is the standard class hierarchy analysis algorithm [32]. Simply put, class-hierarchy analysis finds potential bindings for a method call by considering all of the methods overriding the static method signature at a call site. This algorithm tends to be over-conservative. A more precise algorithm, Rapid Type Analysis [5], could be used to determine potential method bindings. Rapid type analysis can easily be implemented in FEAT at the cost of a small time and space penalty. Further experience with the FEAT tool should determine whether these penalties are warranted.

The relations detected by the analyzer are stored in a database consisting of a hash table. The keys in the hash table are global identifiers for Java elements in the model. The value associated with a key in the hash table is a list of structures comprising a relation name, a range element, and the source code location corresponding to the relation. This structure supports performing a projection operation in a time that, in practice, is only output-sensitive. Since FEAT queries correspond to fragment projections, the execution time for FEAT queries is negligible (less than one second). Likewise, because source code locations corresponding to a relation are stored in the database, viewing the source code for a relation does not incur any perceptible delay.

The static analysis required to extract the model of a program, and the size of the database produced, both impose practical limits on the size of the programs analysable by FEAT. To allow FEAT to work on large programs, it was necessary to introduce a mechanism for users to control the scope of the analysis. We have addressed this issue by defining a concern graph over a set of Java packages. When creating a new concern graph for a project, a user can select from a list of all of the source packages for a project just those packages that should be included in the program model database. Elements declared in packages left out of the analysis can still be viewed and used in FEAT, but their source code is not analyzed. This flexibility allows users to remove basic libraries and other elements that they know are not involved in the concerns they are analyzing, reducing the storage and computation load on the tool.

Because the analyzer performs a single pass through the source code, the time required for model extraction increases linearly with the size of the source code analyzed. Similarly, because there is an approximately constant ratio of model relations per line of code, the space required to store the model also increases linearly with the size of the source code analyzed. We illustrate the time and space cost related to model extraction in the FEAT tool by presenting measurements obtained by loading an increasing number of packages from the jEdit application. Figure 3.13 presents the load times.[4] In the figure, the horizontal axis shows the number of lines of code (LOC) included for analysis.[5] The vertical axis shows the corresponding elapsed time for an initial (i.e., un-cached) model extraction (in seconds). Although model extraction times are subject to many imponderables, such as the effects of multi-threading or virtual memory, loading times follow a marked linear progression, with an origin close to zero and slope of just under 1.1s/kLOC. After an initial model extraction, times for new extractions can be expected to decrease due to the effect of memory caching. For example, we took five samples of the model extraction time for the complete code of jEdit after an initial load. The average time was 57.5s, with a 7% maximum variation.

[4]All times were measured using Eclipse 2.1 on a Windows XP 2002 PC with a 1.8GHz Intel Celeron processor and 512MB of RAM.

[5]All LOC numbers correspond to non-comment, non-blank lines of Java source code.

Contrasted to the 70.8s measured for the initial extraction, the cached extraction presents a 19% reduction in time.

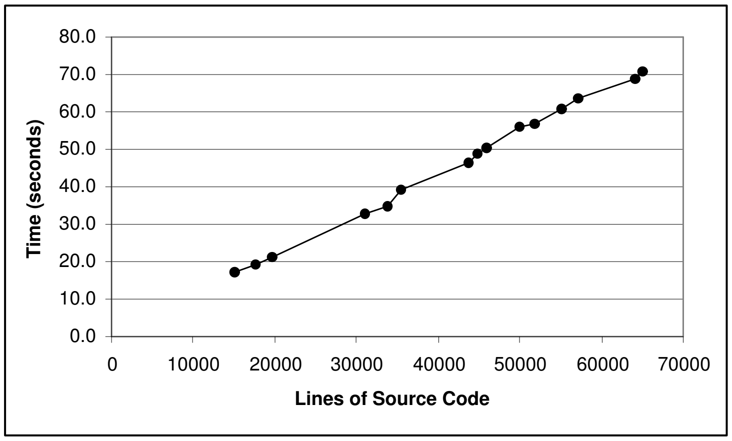


Figure 3.13: Model extraction time

Because it is notoriously difficult to measure the memory consumption of Java programs directly, we describe the size of the model in terms of number of relations. The number of relations is an accurate indication of the size of the model because the size of any relation object is constant: the size of the database is thus determined by the number of relations stored. Figure 3.14 shows the number of relations stored in the model, as a function of lines of source code analyzed. Again, we observe a linear progression, with the origin close to zero, and a slope of about 1.75 relations/LOC.

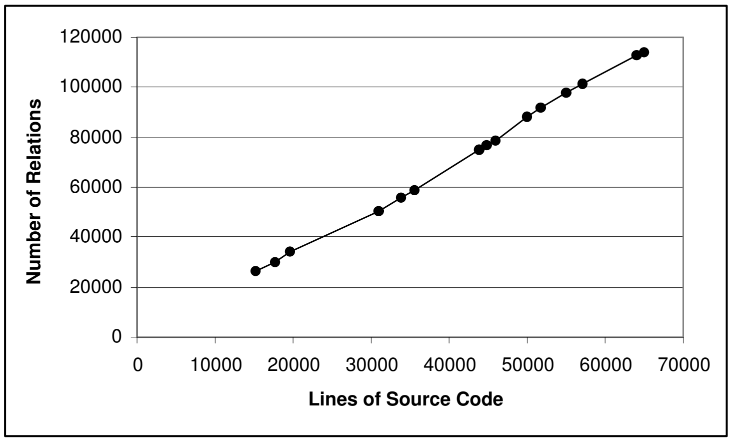


Figure 3.14: Model database size

Finally, to mitigate the cost of model extraction, the FEAT tool updates the model incrementally after an initial extraction. In other words, after an initial extraction, every time a source file is changed, FEAT analyzes the changes and updates only the affected relations in the model database. This technique avoids costly periodic re-extractions of the model.

User Interface

The responsibility of the user interface component is to support the functionalities described in Section 3.2.2. The FEAT user interface is implemented by contributing functionality to the Eclipse platform through the Eclipse extension point mechanism and Application Programming Interface (API) [93]. Two of the main design issues related to the user interface component are the visualization of concerns, and the support for different fragment types.

There exists a duality in the concern graph representation. On one hand, concern graphs are a graph structure of program elements and the relations between them. On the other hand, concern graphs are a recursively-defined hierarchy of concerns and fragments. The graph representation may be better for some tasks, such as analyzing the interactions between two concerns, while the fragment representation is better suited to other tasks, such as moving fragments from one concern to another. We have chosen to present the former (graph) representation to users, and hide the details of the composition of fragments into concerns (except in the Inconsistency View). Hiding fragment composition has the advantage of eliminating the need for users to reason about the complexity of assembling fragments. With the current user interface, the concept of fragments is completely hidden (except in the Inconsistency View). One of the consequences of this choice is that only minimal support is available for tasks that directly involve fragments. For example, it is not possible to move a fragment from one concern to another. We have thus traded flexibility for simplicity in the interface. However, exposing the graph structure of concern graphs to users represents a challenge. Displaying graphs visually has always been fraught with problems of scalability, readability, and layout. For this reason, we have chosen to display concern graphs as a collection of participants and relations. This representation has the additional advantage of being close to the views provided in existing integrated development environments.

Finally, we felt that supporting all of the types of fragments described in Chapter 2 would overly complicate the FEAT tool and impose an unreasonable cognitive load on users. For this reason, we have focused on supporting only the two most useful fragment types: primitive fragments, and fragments with a primitive domain and a universal range. Fragments having another fragment as domain were originally supported, but have been removed from the interface to simplify inconsistency management. Further research should help establish the cost-benefit tradeoff related to the use of complex fragments structures in the definition of concern graphs.

Chapter 4

Validation

The thesis of this dissertation can be decomposed into three claims. A first claim is that a concern graph can help developers perform software evolution tasks more systematically. We will refer to this claim as the **usefulness** claim for concern graphs. A second claim is that a concern graph can be produced cost-effectively during program investigation activities. This claim will be referred to as the **low-cost** claim. Finally, the third claim is that a concern graph can be used to support software evolution on different versions of a system. This will be referred to as the **robustness** claim.

To validate these claims, we have performed a series of five case studies of program evolution using concern graphs. Each case study was designed to investigate specific research questions, focusing on one of the claims above. We refer to each case study with the name of the software system evolved as part of the study. In an initial study, we performed a change task on a small system called AVID to evaluate how useful the concern graph idea was in practice. The AVID study focused on evaluating the usefulness claim. In a second study, we asked a small group of developers to use FEAT to investigate the code for a static analysis tool called Jex in the context of a change task, and to build a concern representation for the code related to the change. The Jex study focused on validating the low-cost claim, and evaluating whether developers not familiar with the concern graph theory could create a concern graph effectively. To investigate issues of scalability related to the technology for supporting concern graphs, and strengthen our low-cost claim, we performed a third case study involving the analysis of a large network provisioning system developed by Redback Networks Canada. The fourth case study involved developers performing a complete change task on the jEdit application described in Section 1.2. In this study, our main focus was the behavior of developers using FEAT during a change task. Results from the jEdit study provide evidence that using concern graphs helps developers perform a change task systematically. Finally, to validate the claim that the concern graph structure is robust enough to capture a concern in different versions of a system, we studied how a concern graph defined on one version of the ArgoUML application could be used on a later version of the same system. The first three case studies were performed using an earlier prototype of the FEAT tool [112], while the jEdit and ArgoUML case studies were performed using FEAT version 2.3.0 (the version described in Chapter 3). Table 4.1 summarizes the claim each study focused on, and Table 4.2 summarizes the characteristics of each study. In Table 4.2, the first two columns list the name of the system evolved as part of each study and its size in lines of code.[1] The third column ("External Participants") states whether the study involved

[1]In this dissertation, unless otherwise stated, all line of code (LOC) figures represent lines of true source

Table 4.1: Claims addressed by the different studies

Study/Thesis claim	1. Usefulness	2. Cost	3. Robustness
1. AVID	$\star$		
2. Jex		$\star$	
3. Redback		$\star$	
4. jEdit	$\star$		
5. ArgoUML			$\star$

Table 4.2: Characteristics of the different studies

Study	System size (LOC)	External Participants	Replication
1. AVID	$> 12\,500$	No	No
2. Jex	$> 57\,000$	Yes	Yes
3. Redback	$\gg 100\,000$	No	No
4. jEdit	$> 64\,500$	Yes	Yes
5. ArgoUML	$> 92\,000$	No	No

participants not directly related to the development of the concern graph approach. Finally, the fourth column ("Replication") states whether more than one case of the evolution of the system was investigated as part of the study.

In the rest of this chapter, we describe and justify our research methods (Section 4.1), and then describe each case study, with the questions it addresses and the results we have obtained. Finally, in Section 4.7, we synthesize the results and discuss the overall validity of the studies.

4.1 Methodology

We have chosen the case study as our validation technique because it is the research method best suited to the explanation of a phenomenon that involves a large number of factors over which only a limited amount of control is available [103, 154]. Program evolution is such a phenomenon, involving developers with diverse backgrounds, large systems, and non-trivial change tasks. In our case, the number and variety of factors affecting the progress and results of realistic evolution tasks preclude a controlled approach. To name only a few examples, the success of a non-trivial evolution task can be influenced by the skill and ability of a developer, the proficiency of a developer with specific techniques such as debugging, the motivation of a developer to succeed in the task, the time of day when the task is performed, the number of pauses taken, the presence or absence of environmental distractions, etc. As a consequence, in our evaluation of the concern graph approach to software evolution, we were more interested in obtaining detailed data that could explain qualitatively *why* our claims were valid or invalid, as opposed to seeking the explanation for causality through statistical inference.

The type of scientific generalization supported by case study research is called *analytic generalization*. Analytic generalization generalizes a phenomenon "to theoretical propositions and not to populations or universes" [154: p.10]. In other words, using the method of analytic generalization, "a previously developed theory is used as a template with which to compare the empirical results of

code, excluding scripts, resource files, comments, and blank lines.

the case study.” [154: p.31]. The analysis of the case studies, as described in the following sections, will thus follow a pattern supporting this analytic generalization. After a brief overview of the study, we formally state the research question the study was intended to answer. For each research question, we then present the theoretical proposition underlying the hypothesis [70]. We then describe the study setting and its results. Then, in each case, we summarize the results in the form of the answer we elicited for each research question. Finally, in each case, we discuss the most important factors affecting the validity of the study. We discuss how the results generalize and the overall validity of the studies in Section 4.7.

4.2 AVID Study

In the first case study, the author of this dissertation took the role of a maintenance programmer to perform a modification to AVID, a Java visualization software system developed at the University of British Columbia [143]. AVID comprises 12 853 non-comment, non-blank lines of code organized in 177 classes and 16 packages. The participant for this case study had no previous exposure to the code of AVID. [2]

4.2.1 Theory

The goal of the AVID study was mostly exploratory, to assess the practical benefits of using concern graphs. The research questions motivating the AVID case study were the following:

1. *Can concern graphs adequately capture the code relevant to a change task?*

2. *How does a concern graph help in performing a software modification?*

Our initial theory for these questions was that:

1. Concern graphs can adequately capture the code relevant to a change task because they capture structural information needed to make the change, and disregard details that are not essential.

2. A concern graph supports a software modification task by providing an uncluttered view of the program elements related to a change task and of the relations between them, so that a developer can easily reason about the change.

The version of the FEAT tool used to perform this and the next two studies is a stand-alone Java application that supports a different concern graph model and a smaller set of features [112]. For the purpose of describing this study, it suffices to mention that the FEAT tool used supports creating a single concern (as opposed to a hierarchy of concerns), supports mostly behavioral queries (see Section 3.1), and does not tolerate inconsistencies. The usage model for the tool used in the study is thus slightly different than the one described in Section 3.2.1 in that users have to first create a concern graph and then modify the code.

[2]The participant was involved in the AVID project as a user of the technology.

4.2.2 Study Design

The study consisted in performing a complete change task for AVID using the FEAT tool. The data collected during the study consists of the modified version of the AVID code, the concern graph produced during the study, and a log of the actions performed in the tool during the study.

The task To visualize the execution of Java programs, the AVID system requires, among other inputs, a file containing summarized information about the events generated during the execution of a Java program [144]. This summary file is generated by a summarizing program. The AVID summarization program takes as input an event trace file and produces a summary file that contains information such as the number of calls and the number of objects allocated or deallocated up to a certain point in the trace file, as determined by some user-defined checkpoint frequency. The summary files also contain information about the age of objects at allocation and deallocation time.

The object age information is voluminous, and experience with the AVID visualizer showed that this information was not always used. Being able to generate and read summary files that did not include this object age information was thus a desirable change for AVID, and we chose it for our first case study.

Finding the concern code In performing this task, the subject used FEAT to discover the concern code that was to be modified, and to save a representation of this code as a memory aid when later performing the change.

The discovery process that was carried out by the subject can be divided into four slightly overlapping phases. A preparatory phase consisted of understanding the application domain and of seeding the concern. This phase did not involve concern graphs or FEAT. A second phase consisted of discovering the part of the code where the writing to the summary files was triggered. A third phase involved understanding and describing the reading and writing mechanism for summary files. A fourth phase consisted of the discovery of a finer implementation detail based on the concern graph that was created, while making the change.

To understand the application domain, the subject spoke briefly with an original developer of the system. This developer explained, at an abstract level, the functioning of the visualizer and the use of summary files. This discussion did not involve viewing source code or explicitly mentioning actual data structures. The only exception is that the original developer mentioned the entry point to the summarizing program, class `PrimarySummarization`. This class was used as a seed to the concern and thus, when the subject started using FEAT, the concern graph consisted solely of this class name.

In the second phase, the subject looked for the major program elements involved in reading and writing to the summary files as a means of gaining an understanding of the format of the files. Using FEAT, loaded with the single entry-point class `PrimarySummarization`, the subject expanded the class and added the `main` method to the concern description. A fan-out query on the `main` method revealed all of the elements used by `main`. These elements consisted of objects being created, and one call to method `summarize` of class `EventSummarizer`. This element was added to the concern graph because it was the only non-library method call. The subject then analyzed the `summarize` method more closely, using both the result of FEAT's fan-out query and the source code viewer. From this information, the subject determined that the only points that could involve writing to the summary file were through calls to `Info.write`, `Summary.write`, and two `store` methods. The subject added these elements to the concern graph. Figure 4.1 shows the concern graph at this point. To produce this concern graph, the subject needed only to find and select the `main`, `summarize`, `write`, and `store` methods. Furthermore, it was only necessary to view the source code of method `summarize`.

In the next phase, the subject discovered the details of the reading and writing protocol for summary files. Specifically, the subject explored the outgoing edges in the program model of the methods discovered in the previous phase to determine what elements actually performed the reading and writing operations, and then explored the incoming edges to analyze the context in which these operations were performed. This phase was more iterative than the first, and included viewing source code through the automatic highlighting feature of FEAT, and exploring dependencies through the query capabilities of FEAT. Using this process, the subject discovered that the code pertaining to the reading and writing of summary files was located in the methods `add`, `read`, and `write` of classes `Info`, `Summary`, `CategoryInfo`, `CategorySummary`, and `CategoryManager`, and a handful of helper methods in the same classes. Once the complete mechanism was discovered, it was possible to determine, by looking at the corresponding source code, that only a subset of the methods identified deal with the reading and writing of object age information. Only these methods were added to the concern graph.

The second and third phases required approximately 90 minutes to complete. The concern graph produced included 3 fields and 18 methods scattered across 7 classes.

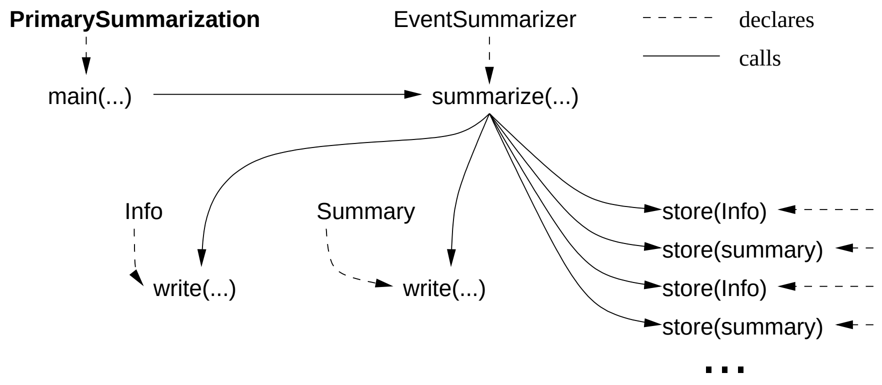


Figure 4.1: Finding the important parts

Making the change To implement the change, the participant visited the source file corresponding to each class in the concern graph once and implemented the changes needed to that class. Of the 18 methods present in the concern graph, 12 had to be modified to implement the required change. Of the remaining six methods, four had object age-related code that did not need to be changed due to specific implementation details. The two other methods were left in the concern graph as structural "bridges" between different parts of the code. For example, method `summarize` (see Figure 4.1) was left in the concern graph as a pointer to the `read` and `write` methods, even if no code in `summarize` actually had to be changed. These methods could have been omitted, as they can be obtained easily with FEAT queries.

To test the change, the modified summarization program was used to generate new summary files both with and without the object age information, and these files were used in visualizing event traces. This allowed the subject to discover that one of the assumptions made about the behavior of the concern was wrong. This assumption was that the first read operation on a summary file would be done through the method `read` of class `Info`. Execution of the program revealed that the first read operation was in fact performed through the `read` method of class `Summary`. To remedy this situation, the subject used the concern graph in a final phase, to find the site of the first read operation to the summary file. The subject iteratively performed fan-in queries, investigating the resulting call sites with the code browser until the context of the calls was determined.

In subsequent testing, the subject successfully used AVID to visualize event traces using the new format of summary files. Making the change and testing it required approximately 150 minutes.

4.2.3 Results

We draw five observations about concern graphs based on the use of FEAT for this change task. The observations are presented from the point of view of the case study subject, an experienced software developer.

Observation 1 *The granularity of the concern graph was sufficient to describe a concern for the purpose of the software change task.*

The subject did not need to consult any other documentation prior to implementing the change. The general behavior of the code learned as part of creating the concern graph was still fresh in memory, and the behavior that was not understood at the time of performing the change could be discovered in minimal time through queries. The concern graph also pointed to the target source code with sufficient accuracy.

Observation 2 *Most of the source code viewed while finding a concern was relevant to the concern.*

An explanation for this observation is that the details of code not related to the concern under investigation were usually discarded at the level of the concern graph model.

Observation 3 *The number of false positives was low.*

In the context of this case study, a false positive is a code element included in the concern graph that did not implement the object age feature. In this case study, only two out of the 19 methods identified in the concern graph were not directly related to the concern. We posit that this low false positive rate is a result of the queries returning elements that are structurally dependent, as compared to text searching tools that can return unrelated items. In this study, the false positives that did occur were methods implementing parts of the object age concern that were not directly impacted by the change.

Observation 4 *The number of false negatives was low.*

The subject made a single pass through the source files to implement the change. Only one method had to be added to the concern graph while performing the change. Our explanation for this observation is that most of the concern code interacts structurally, so the cross-referencing capabilities of FEAT allowed the subject to identify the extent of the concern.

Observation 5 *The program model was not useful in helping to understand highly algorithmic code.*

The subject determined the reading and writing protocol for summary files by reading the source code and the comments of a few specific files. The concern graph was not helpful in understanding this behavior because it did not capture information about the behavior of the concern.

The findings of this study can be summarized in two propositions corresponding to the two research questions.

1. The concern graph provided an adequate representation of the code relevant to the change. Elements not captured by the concern graph did not need to be rediscovered for the developer to complete the task.

2. The concern graph provided good support for documenting the list of methods that needed to be changed. It also provided a quick means to perform additional investigation. It did not provide good support for investigating algorithmic code.

4.2.4 Validity

The internal validity[3] of the AVID study is threatened by potential investigator bias, and by the fact that only one evolution task was considered.

Because the subject in the AVID study was the inventor of the concern graph approach, the results can be expected to reflect a better than average use of the tool. Furthermore, the subject had a vested interest in the success of the study which might have influenced his behavior during the study. To mitigate the potential investigator bias in the analysis of the results, the observations were derived from an analysis of the raw data collected during the study, as opposed to the experience of the subject during the task.

The second threat to the validity of the study is that because only one change task was considered, the results might be accidental due to the nature of the task. In this situation, a small, very focused task would have had the potential of only involving source code that is easily represented through concern graph. To mitigate this risk, we chose instead a task that involved many different types of interactions, scattered in more than 18 methods.

4.3 Jex Study

We performed a second case study to investigate the low-cost claim. Specifically, we were interested in evaluating whether developers unfamiliar with concern graphs and FEAT would be able to build a concern graph for the code related to a change without difficulties or extensive effort.

[3]The test of *internal validity* for a study questions whether the results truly represent “a causal relationship, whereby certain conditions are shown to lead to other conditions, as distinguished from spurious relationships.” [154: p.33]

4.3.1 Theory

The research question guiding the design of the Jex study can be stated as follows:

Can developers build concern graphs effectively while investigating source code in preparation for a software evolution task?

The theory underlying this question is that concern graphs can be built effectively during program investigation activities because they are built from the results of queries usually performed when investigating source code.

4.3.2 Study Design

In this case study, a subject was asked to identify the code contributing to a specified concern in the context of a program change task. We replicated the study three times with three different subjects. In each case, the subject was not asked to perform the change. The target for this task was the Jex system version 1.1 [110, 111]. Jex is a static analysis tool that produces a view of the exception flow in a Java program. Jex is written in Java and consists of 57 152 non-comment, non-blank lines of code organized in 542 classes and 18 packages.

The subjects were asked to identify the code in Jex that handles Java anonymous classes. The context for identifying this code was to change Jex to support a version of the Java language that did not include anonymous classes.

Using FEAT, a concern graph for this concern was produced by the developer of Jex (the author of this dissertation). The elements in this concern graph span 8 classes in 3 different packages. A subset of this concern graph, consisting of one class and one method, was provided as a seed (or starting point) to the subjects of the case study.

The three subjects in this study had diverse backgrounds: one was a senior undergraduate student who had worked in two different companies as part of a co-operative work program, one was a graduate student with previous work experience as a software developer, and one was a developer for a telecommunications company. All of the subjects had some experience with Java, although only one was actively involved in development work with Java at the time of the case study. The subjects had no previous exposure to either the source code of Jex or the FEAT tool.

Prior to performing the task, the subjects completed a 30-minute training session with the FEAT tool, during which they had assistance from the developer of FEAT. The subjects were then asked to produce a description of the anonymous class handling concern that was as complete and as precise as possible. The subjects were instructed to perform the task using only FEAT. In particular, code viewing was to be done only through FEAT's code highlighting function.

The subjects were asked to report the time required to perform the task, their final concern graph, a usage log automatically generated by FEAT, and their confidence in the quality of the result, in terms of estimated percentage of the concern code they had missed. Two additional subjects were involved in prototyping the study. Our experiences with these subjects caused us to adjust the content of the training session to ensure the subjects understood how to use FEAT. The results of the prototype subjects are not included in the results reported.

4.3.3 Results

We analyzed two types of data from the study: the completeness of the concern graph produced (quantitative), and the usage patterns of the subjects (qualitative). We also took into account the time taken by each subject to perform the study. We used the completeness data to verify that the subjects had followed the instructions carefully, and investigated the concern prescribed. We used the usage patterns and time taken to validate the low-cost claim.

For the analysis of the completeness, we used the concern graph produced by the author of Jex as a benchmark. Class and method elements in the concern graphs produced by the version of FEAT used in this study can be marked with a special *all-of* marker if it is deemed by the FEAT user that all of the code for the element is relevant to the concern. One of the 8 classes in the benchmark concern graph was marked as *all-of*. Of the remaining 7 classes, the concern graph includes 1 field and 15 methods: 6 methods are labeled *all-of*; 12 code elements, such as the use of a field, are specified as part of the concern in the remaining 9 methods. Figure 4.2 shows a view of this concern graph. The first level of indentation represents classes. The second level of indentation represents class members, and the third level of indentation represents the uses of class members in method bodies.

```
class JexFile
  all-of method isAnonymous
class Workspace
  all-of method getExceptionFromAnonymousClasses
all-of class AnonymousJexFile
class JexLoader
  all-of method getExceptionsFromAnonymousClasses
  all-of method getTypes
class JexPath
  method main
    calls JexPath.getAnonymousJexFiles
  all-of method getAnonymousJexFiles
class JexFileCollection
  method dump
    calls JexFile.isAnonymous
  method writeJexFiles
    checks AnonymousJexFile
class JexVisitor
  method addExternalNonVirtualCallExceptions
    calls JexFile.isAnonymous
  method addVirtualCallExceptions
    calls JexVisitor.addAnonymousVirtualCallExceptions
  all-of method addAnonymousVirtualCallExceptions
class TypeDeclarationCollectorVisitor
  field aNextAnonymous
  method visitNewObjectExpression
    writes TypeDeclarationCollectorVisitor.aNextAnonymous
  method visitClassDeclaration
    creates AnonymousJexFile
    calls AnonymousJexFile.<init>
    reads TypeDeclarationCollectorVisitor.aNextAnonymous
    writes TypeDeclarationCollectorVisitor.aNextAnonymous
  method <init>
    writes TypeDeclarationCollectorVisitor.aNextAnonymous
  method visitTypeDeclarationStatement
    reads TypeDeclarationCollectorVisitor.aNextAnonymous
```

Figure 4.2: The anonymous class handling concern in Jex

Table 4.3: Concern completeness results

Subject	1	2	3
Classes found (8)	7	6	8
Field found (1)	1	0	0
Methods found (15)	13	7	11
Code elements found (12)	11	3	7
False positives	2	0	0

Table 4.3 shows how many of these elements were identified by the study subjects. Subject 1 found almost all of the concern code in the benchmark, corroborating Observation 4 from the AVID study. The elements missed by this subject were the result of minor inconsistencies in building the concern graph. For example, the participant included the call to method `JexFile.isAnonymous` in method `JexVisitor.addExternalNonVirtualCallExceptions`, but failed to include the declaration of method `isAnonymous` itself in the concern graph. This situation could be avoided automatically if FEAT included the targets of edges in the concern graph.[4] Subjects 2 and 3 missed a higher number of elements. The majority of their false negatives resulted from a failure to see that one field, `aNextAnonymous` of class `TypeDeclarationCollectorVisitor`, was involved in implementing the concern. This field was found by the expert and Subject 1. The expert found the field because, in the source code, the field was referenced close to the call to the creation of an `AnonymousJexFile` object in method `visitClassDeclaration`. Reference to this field was also visible in the results of a fan-out query. Once field `aNextAnonymous` is discovered, a fan-in query on the field returns five out of the seven elements of class `TypeDeclarationCollector-Visitor` related to the concern.

The number of false positives in the concern graphs produced by the subjects was low. Of the three subjects, only one produced a concern graph with false positives: this graph had two false positives which were clients of the functionality described by the concern rather than elements of the concern. This data corroborates Observation 3 from study one. In general, we found the completeness data indicative that the subjects had focused on the right functionality (as opposed to navigating arbitrary structures during the experiment). This increases the validity of the results.

The subjects each produced a concern graph in less than 50 minutes. We find the quantitative results of this case study encouraging because the subjects, who all had minimal training with the concept of concern graphs and the FEAT tool, were able to narrow down, in a short amount of time, an unfamiliar code base of 57 kLOC to a concern graph that captured many of the pertinent parts of the concern.

To validate the low-cost claim, we also analyzed the usage logs collected from the use of FEAT by the subjects. These logs show that approximately 80% of the source code viewed while finding a concern was relevant to the concern (Observation 2). This measure is approximate because viewing an element opens the entire source file. As a result, it is possible to view different elements in the same file. Nevertheless, we interpret this measure as indicative that the subject did not rely on intense code-reading strategies to understand the source code, and instead could rely on the queries provided by FEAT.

[4]Version 2.3.0 of FEAT now does this automatically.

We also found, however, that, as in the first case study, the subjects in this study were unable to use the concern graph to capture system behavior. Moreover, they were unable to use the approach to represent subtle aspects of the structure (Observation 5). For example, even though both subjects 2 and 3 viewed code related to method `JexLoader.getTypes`, neither of these subjects incorporated this method in their concern graph. The `getTypes` method belonged in the concern graph because it was a private method performing specific services for loading anonymous Jex files. To discover this information, subjects had to observe that the caller of the method was part of the concern, and that there was no other caller of the method.

To summarize the results, we found that the three developers involved in the study managed to create a concern graph capturing most of the code relevant to the change task using the structural queries supported by the FEAT tool. We find this result in support of our theory.

4.3.4 Validity

For the Jex study, the principal threat to the overall validity is a threat to its construct validity.[5] To ensure that the data correctly reflects the low cost of building concern graphs, we triangulated [20] three different data sources: the time taken by each subject, the final concern graph produced, and the log of the actions performed by each subject. The concern graph produced by each subject helped us establish that the subject's actions during the study corresponded to the task. The time taken and the analysis of the code investigated by the subjects helped us determine how much time was spent investigating irrelevant code. Redundancy in the interpretation of each data source contributes to increase our confidence in the construct validity of the study.

4.4 Redback Study

To evaluate whether the technology supporting concern graphs scales, we applied FEAT to NSC release 2.1, a large network provisioning code base developed by Redback Networks Canada, Inc.

4.4.1 Theory

The research question of interest for the Redback study was simply whether the concern graph approach scales. Our claim, associated with the general low-cost claim, is that the concern graph approach does scale. We theorize that the approach scales because it is based on a program model that captures the essential elements and relations of a program, as opposed to all of the details of the source code.

4.4.2 Study Design

This study consisted in producing a program model of a large industrial code base with the FEAT tool, to use FEAT to capture existing scattered concerns in the code base, and observing and documenting any issues associated with the scalability of the approach.

[5]The test of *construct validity* questions whether the operational measures used correctly reflect the concept studied.

We applied the FEAT tool to the code of the Redback Canada NSC code base. The NSC code base comprises 233 packages and 1489 classes. It depends on approximately 9 MB of third-party libraries.

4.4.3 Results

The approach taken in the FEAT tool is to load the entire program model into memory. This approach allows users to quickly perform dependency analyses on any parts of a program, and to dynamically reconfigure the environment used to evaluate the queries.

In the case of the NSC code base, it was not possible to load all of the application classes and their dependent classes into the memory available on the analysis machine.[6] The very large size of the NSC code base made it necessary to find a way to selectively restrict the in-memory model of the program. We accomplished this restriction by modifying FEAT to fully load only a user-defined set of classes. Other classes were loaded as stubs that included some information about the class and its members but that did not include the entire bytecode necessary to derive behavioral relations. A consequence of this tradeoff is that any class loaded as a stub could not be queried for dependencies to a program element, except if these dependencies could be detected without the bytecode (e.g., field types, method parameter types). In practice, this approach does not influence the results of the queries if the classes loaded as stubs do not transitively depend on the application classes of interest, which is generally the case with library code and low-level application code. Loading some classes as stubs does not influence the completeness of the class-hierarchy analysis that is performed to determine the potential targets to virtual calls because this analysis requires only method signatures.

To verify that FEAT was operating correctly given these optimizations, we used it to identify the code corresponding to a port to a new error handling framework that had been added in a previous version of NSC. By differencing the code in the versions recorded before and after the change, we were able to determine that the code we identified using FEAT corresponded to the change.

To summarize the results, for the concern graph approach to scale to very large programs, it is necessary to restrict the program model. As a result of this study, we have included a mechanism to restrict a program model loaded in FEAT to classes declared in a set of packages specified by a user of the tool.

4.4.4 Validity

The research question for the Redback study was technical. There are no significant threat to the validity of the results.

4.5 jEdit Study

To strengthen the validation of the usefulness claim established in the AVID case study, we performed a replicated case study of a complete evolution task in jEdit. The change task we used for this case study is the task we have been using as a running example through this dissertation (see Section 1.2).

[6]The machine used had 256MB of memory.

4.5.1 Theory

The jEdit case study focused of validating the claim that concern graphs can help developers perform software evolution tasks more systematically. As such, our investigation is aimed at answering the following research questions.

1. *How* do developers use concerns graphs during program evolution?
2. *Why* is the behavior of developers using concern graphs more (or less) systematic?

The theory underlying these research questions relies on two main hypotheses, which we designate as the *precise investigation* and *precise information capture* hypotheses.

1. **Precise investigation:** By investigating source code following structural relations, and focusing on one concern, developers spend less effort investigating irrelevant information.
2. **Precise capture** When investigating source code, developers use concern graphs as an abstraction to preserve essential knowledge about the different elements in the source code involved in a change, and of the relations between the different elements. Such activities lead to more effective program modifications because the information captured is directly linked to source code, so that code relevant to a change does not have to be re-discovered by navigating through non-relevant source code.

4.5.2 Study Design

The basic design for the jEdit study was to monitor the activities of different subjects performing a complete program evolution task with or without the FEAT tool. Specifically, we replicated the investigation with two subjects using FEAT and two subjects not using FEAT (the control group). We chose the jEdit system for this study because it is large enough to preclude a systematic understanding of the entire code base by the subjects during the time alloted, and because a large system allows us to study a change task that is representative of change tasks in industrial settings. In the rest of this section, we describe the task the subjects had to perform, the process of a replication of the program evolution task, and how the subjects were selected for the study.

The Task The target system for the task was the jEdit text editor (version 4.6-pre6).[7] jEdit is written in Java and the version we used consists of 64 994 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. An overview of this task was presented in Section 1.2. We provide the complete details of the experimental setup here.

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 accessed through 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.

[7]http://www.jedit.org.

The 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 shall 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 shall be saved within the specified autosave frequency.*

5. *When the autosave feature is disabled, the tool shall not 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.*

Understanding the complete set of functionality related to the change task involves reasoning about the use of approximately five fields and 27 methods scattered in 10 classes. The change, as initially performed by the author of this dissertation in preparation for the study, amounted to about 65 lines scattered in six classes.

Study Phases The study was divided into four or five phases, depending on whether a subject used FEAT (FEAT group) or not (control group). To minimize potential investigator bias, each phase was described entirely through written instructions. In any phase, the subject could ask questions, but we established guidelines restricting answers from the investigator to clarifications of the written material.

Eclipse Training Phase To investigate the code and to perform the change, subjects were to use the Eclipse Platform.[8] Eclipse is an open-source integrated development environment for Java. It is a state-of-the-art environment, supporting sophisticated search and cross-reference features, an integrated debugger, a syntax-highlighting editor, etc. Because subjects did not have to be 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.

[8]http://www.eclipse.org.

FEAT Training Phase A subject assigned to the FEAT group was required to complete a training tutorial on the FEAT tool. The training tutorial instructed the subject 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, the subject was asked to experiment freely with the tool. The complete training phase for the FEAT tool was limited to one and a half hour. Before continuing on to the next phase, the subject had to pass a proficiency test, in which the investigator asked the subject to perform various tasks covered in the tutorial.

Program Investigation Phase After all training, a subject was 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 4.5.2, and a set of eight test cases covering the basic requirements. The written material for the 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. A subject assigned to the FEAT group was given these same pointers in the form of two pre-loaded concerns in the FEAT tool, each concern containing one class.

A subject was then given one hour to investigate the code pertaining to the change in preparation to the actual task. A subject was 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). A subject was allowed to take notes in a text file. A subject was 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, we recorded all of the activities of the subjects using the Camtesia screen recording program[9] operating at 5 frames/seconds and a resolution of 1280 x 1024 pixels.

Program Change Phase In this phase, a subject was instructed to implement the requirements as well as possible. A subject was given two hours to implement the change. Use of the debugger was again disallowed. This phase was also recorded using the Camtesia 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 the 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

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

or hindered them in completing their task. The interviews were recorded using the Camtesia screen capture software with an audio input stream, so that the comments of the subjects could be synchronized to code that the subjects identified on the screen.

Subject Selection Subjects for this study 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. Subjects were paid for their time at an hourly rate of 20 CND$. As part of the study data, each subject was asked to state whether he/she had previous experience with Eclipse, and to estimate his/hers programming experience, in terms of number of full-time months of programming experience (in any programming language), and number of months of Java programming experience (with proportional equivalence factors for part-time). The study presented in this section is a refinement over a larger previous study [115], which indicated problems with the usability of the FEAT tool. The problems identified have been corrected prior to undertaking the investigations reported here. Chapter 6 discusses the evolution of the user interface to the FEAT tool in more detail.

4.5.3 Results

The data collected as part of the jEdit study includes the experience of the subjects involved in the study, the time taken to perform the task, the final version of jEdit after the evolution task, screen capture movies of the change investigation and change execution phase for each subject, and the interviews. After collection, each source of data was processed for analysis. To reflect the inaccuracy of the self-reported experience metric for each subject, the number of months of programming experience reported was converted to two broad categories: high, or low, with different intervals for general experience and Java programming experience. The measure of the time taken to execute the change was discarded as invalid for the purpose of our analysis because the subjects were not asked explicitly to optimize the time taken to perform the task. For this reason, we do not report the time measures here as it would mislead the interpretation of the results. The code produced by the subjects as part of the task was inspected for correctness and general quality. The solution for each subject was deemed of high quality if it respected the existing design and implementation decisions implicit in the code of jEdit. The screen capture movies were transcribed into a series of actions performed by the subjects. Transcripts are discussed in more detail below. Finally, each interview was transcribed and analyzed for consistency with the actual actions of each subject. This analysis revealed serious inconsistencies between what the subjects thought they did (as reported in the interview), and what they actually did (as evidenced by the screen capture movies). For this reason, we have also discarded the interviews because we judged them to be an unreliable source of data.

To analyze each case, we have thus focused on the resulting source code and the screen capture movies (and corresponding transcripts) as our main source of data. To ensure that we were comparing subjects of relatively equal ability, we have used the modified version of the jEdit source code to validate that each subject had succeeded in the task. Once this assertion was verified, we used the screen capture movies and transcripts to investigate how the subjects behaved around two types of activities: *information discovery*, and *information use*. Information discovery relates to how developers find important information about the implementation of a concern in source code, and how they capture this information. Information use relates to how developers retrieve information previously discovered, and use it to carry out a program evolution task. According to our theory of precise investigation, investigating the code based on the Java Standard program model (supported by FEAT) in the context of a concern should help developers focus on the code relevant to a concern and avoid the perusal of irrelevant information. According to the precise capture theory, concern graphs should help developers capture information closely related and relevant to the change task, and access this information without having to peruse non-relevant information. To elicit data supporting or invalidating our two theories, our approach was to select a piece of information about the implementation of jEdit that was critical to the implementation of different requirements (independently of the strategy chosen), and to analyze how the subjects discovered and used the information. This analysis revealed that subjects using FEAT used a more streamlined approach to discovering and capturing information about the implementation of a concern, thus supporting both the precise capture and precise discovery theories, and validating the usefulness claim. In the rest of this section, we present the detailed analysis of the behavior of the four subjects which form the basis of the validation of the usefulness claim. We first describe and justify the benchmark information we chose to study, then describe the characteristics of the four subjects and the quality of their modification. Finally, we present the qualitative analysis of the subjects' behavior during the study.

Benchmark As the target of our detailed qualitative analysis of subject behavior, we have chosen the investigation and implementation of requirement 5 in the modification request presented to the subjects (see Section 4.5.2). The correct implementation of requirement 5 by a subject requires, at the very least, the discovery and understanding of a call between methods `load(View,boolean)` and `recoverAutosave(View)` of class `Buffer`. Method `load(View,boolean)` loads a file buffer in memory. If an autosave file for the buffer is detected, it calls method `recoverAutosave(View)` to perform the recovery. The implementation of the requirement involves testing whether the autosave property is enabled, and by-passing the call to `recoverAutosave(View)` if autosave is disabled.

In the context of our analysis, this simple implementation concern presents several desirable characteristics:

- The call to `recoverAutosave(View)` is not located near any code dealing with other issues of the autosave concern. As such, the likelihood of the call being discovered by chance is small; it requires a conscious effort.

- As opposed to other requirements, the call to `recoverAutosave(View)` is not directly related to any member of the two classes given as a seed to the subjects. As such, the information cannot be discovered through a simple query on the initial clues.

- In contrast to the other requirements, requirement 5 involves finding a single and precise point in the program. There exists no ambiguity about whether or not the developers have found the right information.

For these reasons, the behavior of subjects investigating and implementing requirement 5 is likely to be representative of the behavior of the subjects in a real setting.

Subjects In the rest of this section, the FEAT subjects are referred to as F1 and F2, and the control subjects as C1 and C2.

The four subjects are all experienced programmers. Table 4.4 provides a relative evaluation of each subject's characteristics (provided by the subjects using a strict guideline). Data in the Eclipse column indicates the level of proficiency with the Eclipse development environment. A high value indicates that the subject had used Eclipse for real development tasks, whereas a value of low indicates that the subject either has either never used Eclipse, or has only tried it. The experience column indicates the overall programming experience of each subject. A value of low indicates between three and five years of full-time programming experience (or equivalent); a value of high indicates more than five year of experience. The Java column indicates the experience of each subject with the Java language. A value of low indicates less than one year, while a value of high indicates between two and three years of experience. When recruiting subjects for the FEAT group, we looked specifically for subjects with a low level of Java programming experience, so that any relatively better performance may not be correlated with experience with coding in Java.

Table 4.4: Subject Characteristics

Subject	Eclipse	Experience	Java
C1	Low	High	High
C2	Low	Low	High
F1	High	High	Low
F2	High	Low	Low

Solution Quality The modification implemented by all four subjects analyzed passed all of the test cases provided to the subjects. Additional inspection of the source code produced by each subject ensured that the solutions were correct and respected the existing design of jEdit. The solutions differed only in minor and subjective implementation decisions. It is important to note that developer behavior, not implementation quality, is the dependent variable in our analysis; quality is dependent on too many factors to be evaluated directly. As such, we consider that a relatively homogeneous quality of solutions between our subjects, instead of confusing the results, adds to the validity of the study by ensuring the adequacy of skills of the subjects in the study.

Behavior For each subject, we describe how the call between methods `Buffer.load(View, boolean)` and `Buffer.recoverAutosave()` was discovered and used. All of the descriptions are based on the screen capture movies collected during the study. The transcripts of the movies relevant to this analysis are presented in Appendix C. In the descriptions below, the numbers in parentheses refer to the time of the action in the transcript. The letter I indicates a time in the investigation phase, whereas E indicates a times in the execution phase.

Subject C1 Subject C1 traversed the relevant methods three times before recording the information as relevant. In a first pass (I-0:39:59), the subject viewed method `recoverAutosave` while browsing the general structure of the `Buffer` class. This discovery was accidental; the subject did not explicitly record the information, and immediately moved on to the investigation of other methods. In a second pass, the subject viewed both methods of interest while scanning all the accessors of the `Buffer.autosave` field (I-0:45:59). Again, the methods were traversed while the subject was investigating a different concern, and the subject moved on without explicitly recording the information. In a third pass, the subject was explicitly investigating the code relevant to requirement 5. This fact was validated by a comment to this effect written by the subject in a textual notes files at I-0:53:08. At this juncture, the subject decided to scan the `jEdit` class to find information relevant to requirement 5. Then, the subject saw the method `recoverAutosave` in a view showing the results of a previous query. At this point this is completely accidental; the method would not have been shown if a different search had been performed last. Nevertheless, the subject seemed to recognize the method as relevant and attempts to record the information in the notes files. Recording this information took two attempts because the subject did not remember the name of the method properly. This difficulty in recording the information illustrates the need for a representation of concerns code that is directly linked to the code elements.

During the execution phase, the subject viewed the notes (E-0:56:30), presumably to view what should be changed to implement requirement 5. The subject then recalled the `Buffer` class in the editor. Not finding the `recoverAutosave` method, the subject then browsed the `Buffer` class in the Package Explorer, selecting the `recoverAutosave` method (after making a mistake by selecting the wrong method).

These observations show that the investigation is not precise because the subject found the `recoverAutosave` accidentally three times before recording it in a free-form text file. Information capture was also not precise as the subject made typographical errors while recording the name of the method, requiring multiple window switches. We also observe that information capture was not precise because the subject needed to browse the `Buffer` class to find the `recoverAutosave` method, in this case also making a mistake by selecting the wrong element.

Subject C2 Subject C2's discovery of the benchmark information was also characterized by many serendipitous encounters prior to the explicit investigation of the information. Specifically, the subject first viewed `recoverAutosave` while scanning the methods of class `Buffer` (I-0:19:08). The method `load(View)` was found through a lexical search on the keyword "delete", in an investigation unrelated with requirement 5 (I-0:24:47). Method `recoverAutosave` was accessed again while scanning the methods of `Buffer` at I-0:42:12, and for no obvious reason since this action is followed by a separate thread of investigation in `jEdit`. At I-0:43:53, after many traversals of the two benchmark methods, the link between `recoverAutosave` and `load` was finally discovered explicitly through a structural query. At this point, the modified the notes to capture the information "Buffer.load". After more browsing, the subject again modified the notes to include the information "Buffer.recoverAutosave". After a series of unrelated actions, all the callers of method `load` were systematically investigated.

During the execution phase, the subject viewed the notes (presumably to retrieve the name of the methods related to the implementation of requirement 5), and then browsed the `Buffer` class to access the `load` and `recoverAutosave` methods. After a few spurious file switches, the subject returned to `Buffer.java` and implemented requirement 5 in method `recoverAutosave`. The subject then performed a cross-reference query from the editor to identify the caller of `recoverAutosave` (method `load`), and moved on.

Subject C2's discovery and capture of the benchmark information was far from streamlined and precise. The benchmark methods were examined multiple times before the link between them, and their relevance to requirement 5, was identified and recorded explicitly. In particular, method `load` and method `recoverAutosave` were recorded in the subject's notes in two separate events, separated by code browsing. The use of the information was more direct than for C1, with the subject accessing the `load` method immediately after consulting the notes. However, because the information was not linked directly to the source code, accessing the `load` method required browsing irrelevant information in the Package Explorer.

Subject F1 Subject F1 discovered method `load` while systematically investigating all of the methods called by `openFile`, the method in charge of opening a file in the jEdit editor (I-0:41:38). The subject then obtained all of the methods called by `load` through a FEAT query. The results of the query included the call to method `recoverAutosave`. Upon discovery of this call, the subject immediately created a new concern named BUFFER RECOVERY, and added the method call between `load` and `recoverAutosave` to the concern. The subject then investigated the callers and callees of `recoverAutosave`, and then moved on to some other investigation.

Shortly after (I-0:50:20), the subject accessed BUFFER RECOVERY, viewed the information in the concern, and proceeded to investigate all of the callers of `recoverAutosave` and `load`, before investigating a different part of the code.

After the investigation phase, the information contained in the subject's BUFFER RECOVERY concern consisted of the two benchmark methods and of the call between them. In the execution phase, the subject used this information in three cases. In the first case (E-0:22:50), method `load` was viewed but no action is taken, and the subject moved on to other parts of the code. In the second case (E-0:25:09), the subject selected the concern, viewed, alternatively, methods `load` and `recoverAutosave` (presumably to decide where to implement the modification), and then performed the modification in `recoverAutosave`. The modification performed at this point contained a bug; a not operator was missing in a conditional statement.

After some testing revealed the bug, the subject came back to fix the code (E-1:51:08) by accessing the concern and viewing `recoverAutosave`.

The data from subject F1 supports the theory of precise discovery. Using FEAT, the subject discovered a benchmark method while systematically traversing a section of the control flow relevant to loading file buffers. At this point the method was immediately identified as relevant, and added to the concern. The case of subject F1 also shows how concern graph supported the precise capture of the code relevant to the concern. Subject F1 needed to access the benchmark methods twice, once to implement the change, and a second time to fix a bug. In each case the location in the code was accessed directly through the concern description, avoiding the browsing and traversal of irrelevant code.

Subject F2 Subject F2 discovered method `recoverAutosave` while systematically investigating all of the accessors of field `Buffer.autosaveFile` (I-0:23:56). When the `recoverAutosave` method was found, the subject queried FEAT for all of its callers. This revealed method `load`, which the subject investigated. The call between the two benchmark methods was then immediately added to the concern graph, in a concern named RECOVER FROM BACKUP. The information was recalled at one point in the investigation phase (I-0:50:38).

In the execution phase, the subject selected RECOVER FROM BACKUP and accessed and investigated the two benchmark methods. The two benchmark methods were the only methods considered during this episode. After investigating the information in RECOVER FROM BACKUP, the subject performed the change by modifying the `load` method, and then moved on to a different part of the task.

The data from subject F2 also supports the theory of precise discovery, because the subject found the recovery method while systematically investigating the accessors of a field related to the autosave functionality. Precise capture was also attained, as the subject captured only the relation between `load` and `recoverAutosave` in a concern named RECOVER FROM BACKUP, and used only this information when making the change.

Interpretation We now return to the two research questions listed in Section 4.5.1. The first research question for the jEdit study was to determine how developers use concerns graphs during program evolution. The data from our case study shows that subjects F1 and F2 used concern graphs (as embodied in the FEAT tool) as planned. First, they used queries over the program model provided by FEAT to systematically investigate one concern at the time. Second, they recorded precisely the information relevant to the implementation of the autosave recovery in a distinct concern, and used only that information when actually implementing the change.

The second research goal was to determine whether the behavior of developers using concern graphs was more (or less) systematic. Evaluating this research question requires contrasting the behavior of the subjects using FEAT to the one of the control subjects. First, our analysis shows that the investigation performed by the users of FEAT was more precise. Both FEAT subjects found one of the benchmark methods while investigating structural relations to elements relevant to the implementation of the autosave feature; in contrast, both of the control subjects found one of the benchmark methods serendipitously, while browsing the members of the `Buffer` class. This observation supports the theory of precise investigation. Second, the information discovered as part of the investigation was recorded more effectively by the FEAT subjects. The screen capture movies show the control subjects recording information about the task by writing the name of elements in a textual file, a process requiring multiple view switches, and, in the case of C1, the correction of an error. During the execution phase, all four subjects found it useful to access the information they had recorded about the interactions between the benchmark methods to find the location in the code where to implement the change. Again, the movies for the FEAT subjects show a streamlined process when accessing the captured information, with the subjects selecting the relevant concern in the Concern Graph view, and accessing one of the benchmark methods. On the other hand, subjects in the control group needed to view their notes, then browse many unrelated elements in the Package Explorer to find the benchmark methods. In one case, the subject ended up selecting the wrong element. This observation supports the theory of precise capture.

4.5.4 Validity

To limit the threats to the construct validity of the study, our analysis relies on the basic transcripts, and sometimes directly on the actual screen capture movies. This way, minimal divergence is introduced between measures of the subject's behavior and their actual behavior.

The internal validity of our study is threatened by the possibility that the success level and the behavior of a subject is determined by a different, competing factor, such as prior knowledge, proficiency with the development environment, or 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 subjects' 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 answer questions only about the features of the tools covered in the tutorial, and was not to provide any comment about the task.

Finally, the internal validity of the results of the study are also threatened by our selection of a specific requirement for detailed analysis. To mitigate the influence of this decision on the result, we picked the requirement of which we considered the analysis to be the least likely to be influenced by the experimental procedures (see the Benchmark paragraph of Section 4.5.3 for details).

4.6 ArgoUML Study

To validate the robustness claim for concern graphs, we performed a study of the evolution of a large system on which concern graphs were defined. As the target application for this study, we chose ArgoUML, a tool for producing diagrams in the Unified Modeling Language [104, 107, 108]. ArgoUML is developed in Java and consists of between 92 and 100kLOC, depending on the version considered. The code base, revision history, and bug database of ArgoUML are publicly available.

4.6.1 Theory

The questions we investigated in the ArgoUML study is *whether a concern graph can represent the implementation of a concern in two different versions of a system*.

Our theory underlying this research question is that a concern graph can represent concerns in different version of a system because:

1. The fragment structure representing the implementation of concerns in source code captures an abstraction of the interaction between program elements, as opposed to syntactic details of the source code. As such, the concern graph structure is tolerant to minor changes in the source code.

2. The concern graph structure supports the detection of inconsistencies with a code base. Inconsistent fragments can be repaired to reflect the new version of the code base, or used as a starting point to investigate the discrepancies between a concern graph and the source code.

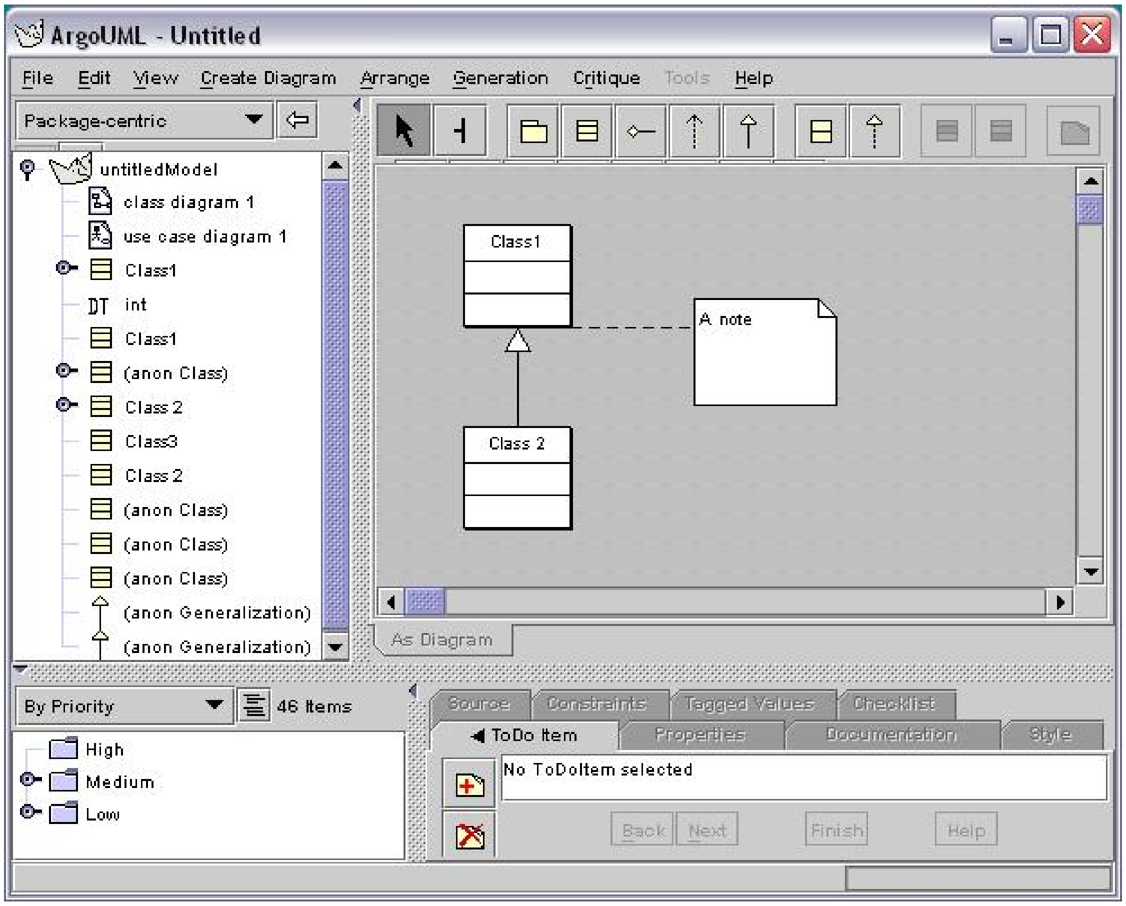
4.6.2 Study Design

The study consisted in the author of this dissertation creating a concern graph capturing the code relevant to the correction of a bug identified for version 0.11.4 of the system, and then loading the concern graph on version 0.13.4 of the system. The data collected during the study consists of the concern graph corresponding to the bug report chosen as a case.

The case The case we focus on in this study is the evolution of the code base to address bug 1209 in the ArgoUML bug database. This bug was identified is version 0.11.3 but was not addressed until version 0.13.1. The bug, as stated in the bug report, is as follows:

> Why can you only add comments/notes on class/state/activity diagrams? According to the uml spec you must be able to add them to all model elements so you must be able to add comments to all diagrams too.

This bug report refers to the possibility of attaching a notes (or comments) box to different objects in UML diagrams. In version 0.11.4 of ArgoUML, it is only possible to attach notes to objects in class, state, or activity diagrams. For other diagrams, such as interaction diagrams, it is not possible to add notes objects. Fixing this bug thus requires modifying the code of ArgoUML to support adding notes to all diagram types. Figure 4.3 shows the window of version 0.11.4 of the ArgoUML tool. The figure shows a simple class diagram, with one of the classes annotated with a notes object. The rightmost icon in the application's tool bar is used to add notes to objects in the diagram. For any diagram type except class, state, or activity diagrams, this button is not visible, and the function is unavailable to users.



No ToDoItem selected

Figure 4.3: The ArgoUML application

Table 4.5: Concern graph for the ArgoUML study

Concern	Classes	Description
ADDING NOTES	26	The root concern (concern graph) for the task.
CREATE NOTE IN MODEL	5	The code to add a note object to the internal UML model.
NOTE CREATION	9	The widget code supporting the creation of a new note object.
ADD ACTION TO TOOLBAR	5	Sub-concern of NOTE CREATION. Code to add the notes button to the toolbar.
DIAGRAM HIERARCHY	9	Classes implementing the different UML diagrams.
CREATE NOTE IN UI	9	Code to display the notes object on the diagram.

4.6.3 Results

Without first looking at the code of version 0.13.4, the subject (the author of this dissertation) created a concern graph capturing the code which seemed relevant to the evolution task in a concern graph. The resulting concern graph, named ADDING NOTES, comprised five sub-concerns and 41 fragments, totaling three fields, 33 methods, and 26 classes (including eight library classes), scattered in 16 different packages. Table 4.5 summarizes the five sub-concerns. The table displays, for each concern, the name of the concern, the number of classes involved in its implementation, and a short description of the concern. The ADDING NOTES concern graph was created by the subject of the study by performing queries in the FEAT tool.

After creating the concern graph on version 0.11.4, the concern graph was loaded on version 0.13.4. Version 0.13.4 implements a great number of changes, including the fix to bug 1209. It is approximately 8kLOC larger than version 0.11.4. To measure how much of the code of ArgoUML relevant to our concern had actually changed between versions 0.11.4 and version 0.13.4, we applied the code comparison feature of Eclipse on the two versions. The code comparison feature of Eclipse compares two files using an algorithm similar to the UNIX `diff` utility [59], returning the differences between unmatched subsequences. Table 4.6 reports the differences between versions 0.11.4 and 0.13.4 for all 18 source (i.e., non-library) classes involved in the concern. For each class (first column), the table lists the number of unmatched sequences between the two version of the source code for the class (second column), the total number of lines in unmatched sequence for version 0.13.4) (third column), the total size of the file in version 0.13.4 (fourth column), and the ratio of the third to the fourth columns (in percentage).

These figures illustrate the amount of change between the two versions. All the classes considered as part of the concern except one (`FigEdgeNote`) changed. Furthermore, most of the changes involved multiple modifications, with half of the classes presenting modifications spanning more than 50% of the lines of code in the class. Given the purely lexical comparison performed by the compare feature of Eclipse, these figures should be understood as representing a pessimistic quantification of the amount of change between the two versions. Nevertheless, the analysis of the differences shows that almost all of the classes in the concern graph were touched. As such, this case represents a good test for the robustness of concern graphs.

Upon loading ADDING NOTES on version 0.13.4, seven fragments were detected as inconsistent (out of 41). Six of the inconsistencies were in ADD ACTION TO TOOLBAR and one was in

Table 4.6: Differences between classes of versions 0.11.4 and 0.13.4 of ArgoUML

Class	Changes	Change Size	Total Size	% Change
AbstractUMLFactory	5	19	77	25
CoreFactory	85	1570	1820	86
ArgoDiagram	12	48	166	29
ProjectBrowser	105	593	789	75
UMLActivityDiagram	15	50	219	23
UML CollaborationDiagram	27	188	259	73
UMLDeploymentDiagram	30	162	214	76
UMLSequenceDiagram	40	156	224	70
UMLStateDiagram	37	179	261	69
FigClass	153	956	1149	83
FigComment	60	214	488	44
FigEdgeNote	0	0	105	0
FigInterface	112	646	789	82
UMLClassDiagram	32	131	197	67
UMLDiagram	13	166	283	59
FigUseCase	163	284	1394	20
UMLUseCaseDiagram	40	127	327	39
ActionAddNote	6	10	235	2

NOTE CREATION. All of the other concerns remained consistent in the new version of ArgoUML. We now discuss each inconsistency, how we repaired it, and the information we could determine from the inconsistent fragment.

The first inconsistent fragment we considered is the primitive fragment:

```
UMLActivityDiagram.initToolBar() calling ToolBar.add(Action)
```

Selecting this inconsistency in the Inconsistency View revealed that method `initToolBar()` of class `UMLActivityDiagram` did not exist. We then queried FEAT for all the callers of `ToolBar.-add(Action)`, to see whether the method had been renamed. Instead, the query revealed that `ToolBar.add(Action)` was called by method `initToolBar()` of class `UMLDiagram`. In other words, the caller was replaced by a method of the same name in a different class. We then queried FEAT for the subclasses of `UMLDiagram`, which revealed, among others, the class declaring the initial caller of `ToolBar.add(Action)`, class `UMLActivityDiagram`. Hence, with two queries using the inconsistency as a starting point, we established that, in version 0.13.4, method `initToolBar()` had been moved from class `UMLActivityDiagram` to its superclass, class `UMLDiagram`. To repair this inconsistency, we replaced the inconsistent fragment by the call to `ToolBar.add(Action)` from `UMLDiagram.initToolBar()`.

The next five inconsistencies we considered were the fragments:

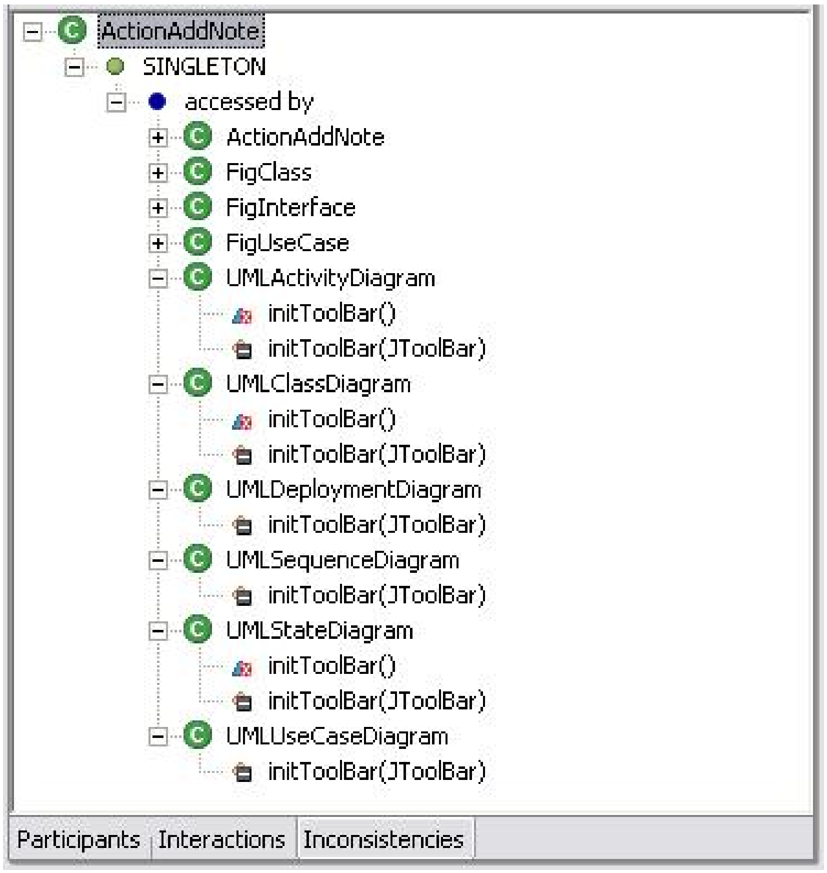
```
UMLClassDiagram.initToolBar() calling ToolBar.add(Action)
UMLStateDiagram.initToolBar() calling ToolBar.add(Action)
UMLActivityDiagram.initToolBar() accessing Diagram._toolBar)
Diagram._toolBar accessed by UMLClassDiagram.initToolBar()
Diagram._toolBar accessed by UMLStateDiagram.initToolBar()
```

These five inconsistencies were also caused by the move of method `initToolBar()` to the superclass. The five inconsistencies were found and repaired in a way identical to the first one.

The last inconsistency was a fragment with a universal range:

```
ActionAddNote.SINGLETON accessed by ALL
```

Displaying this inconsistent fragment in the inconsistency view revealed that the inconsistency was caused by two separate changes. Figure 4.4 shows the fragment above as it appears in the fragment viewer of the FEAT Inconsistency View.



```
ActionAddNote
  SINGLETON
    accessed by
      ActionAddNote
      FigClass
      FigInterface
      FigUseCase
      UMLActivityDiagram
        initToolBar()
        initToolBar(JToolBar)
      UMLClassDiagram
        initToolBar()
        initToolBar(JToolBar)
      UMLDeploymentDiagram
        initToolBar(JToolBar)
      UMLSequenceDiagram
        initToolBar(JToolBar)
      UMLStateDiagram
        initToolBar()
        initToolBar(JToolBar)
      UMLUseCaseDiagram
        initToolBar(JToolBar)
```

Figure 4.4: Representation of the inconsistent fragment `ActionAddNote.SINGLETON accessed by ALL` in the fragment viewer of the FEAT Inconsistency View

1. In three classes (`UMLActivityDiagram`, `UMLClassDiagram`, and `UMLStateDiagram`), the concern graph recorded that the `SINGLETON` field was accessed by method `initToolBar()` of each respective class, which did not exist (represented by the red "X" icon). Also, for each of these cases, the concern graph did not include an actual access to the field by method `initToolBar(JToolBar)` (represented by a minus icon). Clearly, this inconsistency was caused by the modification of the `initToolBar()` method to include a parameter in its signature.

2. In three other classes (`UMLDeploymentDiagram`, `UMLSequenceDiagram`, and `UMLUse-CaseDiagram`), the concern graph was missing an access to field `SINGLETON` in method `initToolBar(JToolBar)` (represented by a minus icon). Clearly, this inconsistency resulted from fixing the bug to support the creation of notes objects in all of the diagrams supported by the tool.

To fix this last inconsistency, we used the automatic repair feature of FEAT, which synchronizes the concern graph with the code according to the algorithm of Section 2.3.2.

To complete this case study, we show an example of a fragment which remained consistent in the face of extensive change to a method. In the concern graph, sub-concern CREATE NOTE IN MODEL contained the fragment:

```
AbstractUmlModelFactory.initialize(Object) calling
    MBase.addMElementListener(MElementListener)
```

The source code for method `AbstractUmlModelFactory.initialize(Object)` in version 0.11.4 is shown in Figure 4.5. In this version of the method, the source code mapping to the fragment is the statement on line 6.

The source code for the same method method in version 0.13.4 is shown in Figure 4.6. In this version of the method, the source code mapping to the fragment corresponds to the statements on lines 12 and 13. As this example shows, capturing the essence of a concern in terms of structural dependencies allows us to preserve the intent of a concern in the face of changing source code. In this example the intent is the call to `addMElementListener`. This intent is captured in the concern graph and can be mapped to the corresponding source in two different versions, even though the corresponding code was modified from a single method call to two calls that take two different objects as parameters.

Returning to our research question for this case study, the results of loading a concern graph created in version 0.11.4 of ArgoUML into a different version, 0.13.4, showed that the concern graph was robust in the face of change. Specifically, although almost all of the classes involved in the concern changed extensively, most concerns in the concern graph remained completely consistent. Inconsistencies stemming from a refactoring of a method, and from the modification of the bug, were corrected easily. Additionally, this case study has shown that the FEAT queries on inconsistent fragments were useful in identifying a refactoring in the code.

4.6.4 Validity

The main threats to the validity of the ArgoUML study are investigator bias in choosing the concern to create, and the creation of the concern graph. To mitigate these factors, we used a modification of ArgoUML related to an actual bug as a case, as opposed to investigating an arbitrary concern in the code base. Second, the concern to investigate was selected before any investigation of the source code was performed. It was thus not possible for the investigator to select a concern that would have a good chance of evolving well. Additionally, the source code for the concern in version 0.13.4 was not examined until the concern graph on version 0.11.4 was completed. As such, it was impossible for the investigator to know in advance how stable the concern graph would be. Construct validity is not an issue in this study since no surrogate measure is used.

```java
protected void initialize(Object o)
{
   logger.debug("initialize(" + o + ")");
   if( o instanceof MBase )
   {
      ((MBase)o).addMElementListener(UmlModelListener.getInstance());
      if( ((MBase)o).getUUID() == null )
      {
         ((MBase)o).setUUID(UUIDManager.SINGLETON.getNewUUID());
      }
   }
}
```

Figure 4.5: Method `AbstractUmlModelFactory.initialize(Object)` in ArgoUML version 0.11.4

```java
protected void initialize(Object o)
{
   logger.debug("initialize(" + o + ")");
   if( o instanceof MBase )
   {
      if( ((MBase)o).getUUID() == null )
      {
         ((MBase)o).setUUID(UUIDManager.SINGLETON.getNewUUID());
      }
      // next two objects are the ONLY two objects that need to listen
      // to all modelevents.
      ((MBase)o).addMElementListener(UmlModelEventPump.getPump());
      ((MBase)o).addMElementListener(UmlModelListener.getInstance());
      Set couples = UmlModelEventPump.getPump().
         getInterestedListeners(o.getClass());
      Iterator it = couples.iterator();
      while( it.hasNext() )
      {
         UmlModelEventPump.ListenerEventName couple =
            (UmlModelEventPump.ListenerEventName)it.next();
         UmlModelEventPump.getPump().
            removeModelEventListener(couple.getListener(),
                                     (MBase)o, couple.getEventName());
         UmlModelEventPump.getPump().
            addModelEventListener(couple.getListener(),
                                  (MBase)o, couple.getEventName());
      }
   }
}
```

Figure 4.6: Method `AbstractUmlModelFactory.initialize(Object)` in ArgoUML version 0.13.4

4.7 Summary

In this chapter, we have described five case studies conducted to evaluate the three important claims of our research hypothesis: that concern graphs can help developers performing change tasks, that concern graphs are inexpensive to create, and that concern graphs are robust enough to describe a concern in more than one version of a system. The studies we performed to validate these claims were all based on the evolution of existing systems that range in size from 12.5 to over 100kLOC.

Table 4.7: Overlap between studies and claims

Study/Thesis claim	1. Usefulness	2. Cost	3. Robustness
1. AVID	$\star$	$-$	
2. Jex	$-$	$\star$	
3. Redback	$-$	$\star$	
4. jEdit	$\star$	$-$	
5. ArgoUML	$-$	$-$	$\star$

Each of the case studies focused on validating a specific claim. As part of the AVID and jEdit studies, we investigated the behavior of developers performing a complete change task with (and without) the support of concern graphs, and showed that concern graphs helped in program modification tasks by supporting a more precise investigation of the code, and by supporting a precise capture of the information related to the change. As part of the Jex study, by investigating how developers with a minimal training with FEAT produced a concern graph describing the code relevant to a change task, we showed that concern graphs could be created without difficulties during program investigation activities. As part of the Redback study, we determined that the concern graph approach could scale given reasonable trade-offs in the construction of the underlying program model. Finally, through the investigation of the evolution of the ArgoUML system, we showed how a concern graph was robust enough to describe concern code in two versions of a system, even though the later version had been subjected to extensive modifications.

Taken separately, each study presents an incomplete picture of the use of concern graphs. In each case, we have made concessions to the necessities of practical empirical investigation involving a prototypical tool. In choosing to validate our approach using multiple case studies, our goal was not only to give structure to the validation, but also to gain experience with the use of concern graphs in different circumstances. Although each study focused primarily on a single claim, the data collected often corroborates and strenghtens claims that were the focus of another study. For example, although the primary focus of tthe jEdit study was the usefulness claim, the data collected showed that subjects using FEAT did not expend any significant effort building a concern graph. This observation contributes to the validation of the low-cost claim. Thus, although the discussion in this chapter was organized along the lines of specific questions, the validation of the concern graph approach should be construed as repeated experiences with the approach in different circumstances, and with different developers performing different tasks on different systems. The overlap in data validating the different claims, the lack of obvious contradictions between studies, and the variety of systems and tasks studied contributes to the generalizability of our results in similar circumstances. Table 4.7 recapitulates the focus of each study (represented with the symbol $\star$), and shows claims for which there exist some secondary validation not explicitly addressed by the study (represented with the symbol $-$).

Finally, conducting the case studies also allowed us to make many observations and to raise many questions that did not fall within the strict framework of the validation. Important issues observed during the studies or explicitly raised by the study subjects were recorded, and are discussed in Chapter 6.

Chapter 5

Automating Concern Graph Creation

Using the tool support for concerns graphs described in Chapter 3, developers must make conscious and planned decisions about what to include and reject from a concern graph. Although we have shown, in Chapter 4, that this activity requires only a minimal level of effort from developers, it can present difficulties in the case of inexperienced or improperly trained developers, or in the case of developers facing intense time pressures. To reduce the cost of producing concern graphs, we have developed a technique to automatically infer basic concern graphs from program investigation activities [113].

The concern inferencing technique we developed extracts a user-specified number of elements from all of the elements considered during a program investigation. It then groups those 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 the parts of the source code a developer investigated during a program investigation session, and on how the developer moved between different pieces of source code during the session. The algorithm requires as input a program investigation transcript obtained by recording, in sequence, every change in the source code 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 elements were accessed, and whether there exists in the code structural relationships between the elements examined. For a specified number of program elements which can be set arbitrarily, 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 algorithm presented here serves as a basic proof of concept of the feasibility of automating the building of concern graph descriptions. The algorithm was developed heuristically by experimenting with the use of a structure, called a navigation graph [115], that describes the paths taken by developers as they investigate the source code of a program. We expect that additional work on the development of algorithms to automate concern graph construction could produce even more precise results. In Section 6.7.1, we discuss in more details our plans for future research on this subject.

In the rest of this chapter, we describe the format of the investigation transcripts we use as the input to our algorithm (Section 5.1), we describe our inference algorithm (Section 5.2), and 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 (Section 5.3).

5.1 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 method declarations and, in some cases, field declarations. Other elements normally present in source code, such as class declarations and comments, are not considered. We chose this approach because the concern graphs 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 5.1 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 5.1: Example investigation transcript

5.2 Inference Algorithm

Given a program investigation transcript, our aim is to automatically extract potential concern graphs. The concern graphs produced by our algorithm consist in a single concern containing a list of program elements specified as primitive fragments using the identity relation (see Section 2.2.2). In other words, the concern graphs produced by our inference algorithms are lists of program elements. For simplicity, in the rest of this chapter, when it is clear from the context whether we are referring to a user-level concern or a concern graph, we shall use the term concern interchangeably. We propose an algorithm that can generate concern (graphs) 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.

5.2.1 Calculating Probabilities

As we mentioned in Section 5.1, 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, 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.

```
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 5.2: Calculating probabilities

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 5.2, 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 5.1 are: $p(\text{F29}) = \frac{1}{8} = 0.125$, $p(\text{F30}) = \frac{6}{8} = 0.75$, $p(\text{F31}) = \frac{1}{8} = 0.125$.

5.2.2 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 5.3.

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).

5.2.3 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 concerns graphs is produced, a developer can choose which ones represent the implementation of actual and useful concerns considered during the program investigation, and name and save the useful concern graphs for later use.

```
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 $c_{k+1} = \mathbf{C}$ then
        $m_{i,j} = m_{i,j} \cdot \beta_C$
      else if $c_{k+1} = \mathbf{R}$ then
        $m_{i,j} = m_{i,j} \cdot \beta_R$
      else if $c_{k+1} = \mathbf{L}$ then
        $m_{i,j} = m_{i,j} \cdot \beta_L$
      else if $c_{k+1} = \mathbf{K}$ then
        $m_{i,j} = m_{i,j} \cdot \beta_K$
      else if $c_{k+1} = \mathbf{S}$ 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 5.3: Calculating correlation metrics

5.3 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 [93]. 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.

5.3.1 Implementation Status

To obtain the results described in this chapter, we 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,

Table 5.1: 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

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).

5.3.2 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 5.1 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 $\beta_2 = 0$.

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.

5.3.3 Studies

The first set of data is taken from the jEdit case study described in Section 4.5. As a brief overview, subjects taking part in the study 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: C1 and C2 (described in Section 4.5), and an additional subject from an additional replication [115], referred to as C3. All of these subjects were part of the control group of the study, and performed the change task without the help of the FEAT tool. 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.

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[1], 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[2] 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 5.1.

5.3.4 Results

Table 5.2 describes the size of the transcripts produced by 60 minutes of investigation (subjects C1, C2, and C3) 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.

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

Table 5.2: Characteristics of transcripts

Subject	Nb. Events	Nb. Elements
C1	123	71
C2	175	102
C3	204	105
J1	260	200
J2	142	152

Table 5.3: Results for Subject C1

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			

Using $\alpha = 5$ as our confidence parameter, we applied each of the configurations described in Table 5.1 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 the different concerns as sets of elements.[3] 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 5.3.5.

For subject C1 (Table 5.3) 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 5.3.3 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 graph. The important relations were identified by most of the configurations. The most successful configuration

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

Table 5.4: Results for Subject C2

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 5.5: Results for Subject C3

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			

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 C2 (Table 5.4) the five configurations produced more homogeneous results than C1: 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 C2, the most useful configurations are 1 (basic) and 3 (no structure), as in the case of C1.

For subject C3 (Table 5.5) 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 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 C1 and C2.

In the case of the jHotDraw study, for subject J1 (Table 5.6) the five configurations produced

Table 5.6: 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 5.7: 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	

12 different concerns involving 17 different elements. Concern 1 is a subset of the interactions relevant to the concept **command menu** identified in Section 5.3.3. 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 5.7) 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.

5.3.5 Observations

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 (C1, C2, C3, 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 (C1 and C2), 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 C1 and C2 is that both of these subjects were very organized in their program investigation, investigating elements that were related in the first place [115]. 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 5.2.1). 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 C1, C2, and C3 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.

5.4 Summary

In this chapter, 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 positives (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 the organization of the program investigation activities: Broad and disorganized searches produced vague and incomplete concerns, while more focused program investigation typically yielded 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 can be fully automated, it is possible to easily and rapidly generate descriptions for different concerns developers investigate in source code. These concern graphs can then be used as supporting documentation during program evolution tasks, as a basis to plan refactorings [43, 95], and potentially to help port a system to an aspect-oriented language [69].

Chapter 6

Discussion

In this chapter, we describe the main issues that arose during the development and investigation of the concern graph approach, we summarize our views on the potential impact of concern graphs on the process of software maintenance, and we present a plan for future research involving concern graphs.

6.1 The Development and Evaluation of the FEAT Tool

The technology supporting concerns graphs was developed in multiple phases over approximately two years. The first prototype of FEAT was developed as a stand-alone Java application [112]. It was released for public download with a University of British Columbia End-User License Agreement as version 1.9.1 in December 2001. This first prototype differed primarily from the tool described in Chapter 3 in five ways.

- The graphical user interface consisted in only two types of windows: an abstract view of the code, comprising participants and relations in a concern graph, and a code viewer capable of highlighting the code corresponding to a relation. Executing a query produced a new window containing the results of the query.
- It was not possible to create more than one concern.
- It was not possible to add multiple elements or relations at once in a concern.
- The program analyses supported by the tool relied on bytecode analysis, which required loading each Java class file in a program into main memory.
- The tool did not tolerate inconsistencies between a concern graph and the source code.

We used this version to carry out the first three case studies described in Chapter 4. Informal evaluation of the tool was also performed by users in the Software Practices Laboratory at the University of British Columbia, by students enrolled in a graduate-level software engineering course in the Department of Computer Science at the University of British Columbia, by researchers at IBM's T.J Watson Research Center, and by inventors of different concern-finding tools [82]. These early experiences produced a wealth of feedback supporting the improvement of the approach. In particular, the initial evaluation of FEAT showed that:

- Support for defining more than one concern was desirable;
- Support for specifying the entire results of a query as part of a concern was desirable;
- The creation of new windows to display results was confusing to users.

These and many other observations motivated a complete review of not only the tool, but also the concern graph model. Following the experiences with the first FEAT prototype, we evolved the concern graph model described in [112] into the model described in Chapter 2. This new model addresses the most important issues noted in the initial investigation phase. Specifically, it allows the definition of multiple concerns in one view, it allows the addition of nested query results to a concern graph as a single unit, and it tolerates inconsistencies between a concern graph and a model. To leverage from the many benefits of integrating the tool in a state-of-the-art software development environment, support for the new model was completely re-implemented as a plug-in for the Eclipse platform. The first version of this plug-in (2.1.4) was released for public download in December 2002. This early version supported most of the features described in Chapter 3, except that it did not tolerate inconsistencies between a concern graph and the source code. It also still relied on bytecode analysis to produce the program model.

To test whether the FEAT Eclipse plug-in provided the necessary support for concern graphs, we conducted a preliminary study of program evolution with eight programmers, four of which were required to use the FEAT tool, and four of which were required to perform the evolution task using only the features of the Eclipse platform. This study is described in detail in a separate report [115]. The preliminary jEdit study uncovered two important issues with the new implementation of the FEAT tool. First, the more sophisticated graphical user interface required additional training for developers to benefit from the tool. The issue of training in FEAT is discussed in detail in Section 6.2. Second, the tight integration of FEAT in the development life-cycle, including support for alternatively building a concern graph and modifying the code, was essential. We thus further improved the FEAT tool and its documentation to address these problems. This development resulted version 2.2.1, released in April 2003. The version of FEAT described in Chapter 3 only implements minor improvements to version 2.2.1. With all of the critical issues addressed, we replicated the original study with two additional developers using FEAT. This last evaluation is described in Section 4.5. Because the issues with the original study were limited to the use of the FEAT tool, we did not replicate the study with additional control participants. Instead, we used the two most successful control participants in the original study to contrast with the FEAT participants in the final phase of the study.

6.2 Training and the Use of FEAT

Our experience evaluating the FEAT tool with many different users has helped us identify the important factors that influence its effectiveness. Chief among these factors is the level of proficiency achieved by users of the tool. Achieving a good level of proficiency requires proper training. We have observed that the effectiveness of the training provided to FEAT users is influenced by three overlapping factors: prior exposure to the concept of separation of concerns, experience with program analysis and cross-reference queries, and experience with the use of software development environments. To address these issues, we have ensured that our training material covered these three areas. In particular, the step-by-step tutorial provided to the subjects in the jEdit study comprised:

1. An introductory section motivating, with examples, the need for proper modularization and the concepts of separation of concerns;
2. A section describing how to perform queries and a list of the semantics of all the queries supported by FEAT;
3. Instructions on how to use the basic features of Eclipse, such as performing searches and using a code browser.

After experimenting with a FEAT training session of 60 minutes, we determined that this amount of time was insufficient, and increased the training time to 90 minutes for the two final replications of the jEdit study. After this amount of training, both of the FEAT subjects involved in the jEdit study were able to use the tool properly. A time of 90 minutes thus constitutes a good indication of the minimum effort required to use the FEAT tool effectively.

6.3 Capturing System Behavior with Concern Graphs

When analyzing the results of the AVID and jEdit studies, we observed that the code relevant to a concern sometimes included complex program behavior. For example, to modify the AVID system, the developer needed to consider a constraint on the order of calls to a method. As an other example, when modifying the jEdit system, the developers needed to understand code managing the state of a buffer. The subject in the AVID study (Section 4.2), and most of the subjects who took part in replications of the jEdit study (Section 4.5), failed to properly understand some complex behavior of the system. The program model extracted by FEAT does not support the investigation and capture of this kind of behavioral information. This observation raises two important questions. First, can concern graphs provide any help for complex cases? Second, should more support be provided? In answer to the first question, the case studies have shown that concern graphs are helpful to developers because they provide a means to store a list of program elements that can act as anchors and provide context when investigating complex code. In other words, although concern graphs cannot explicitly capture complex interactions, they can point to the places where such interactions occur, and, through concern names, provide some information about the context in which they occur. Evidence of this type of support is found in both the AVID and jEdit studies. For example, in the jEdit study, both subjects, having realized that some part of a concern was not well understood, used the concern graph to return precisely to the point where further investigation was required. Concern graphs thus provide some minimal support for understanding complex code. There remain the question of whether additional support should be available. At first glance, we can identify three potential ways to provide additional support for fine-grained program investigation and capture: changes to the model to add ordering information, use of a finer-grained model, and support for attaching free-form comments to elements in a concern graph. All these options have important associated costs. First, changes to the model to accommodate ordering, or a finer-grained model, both imply a larger database to store the model, slower analyses, and additional user-interface support to deal with the increased flexibility. Free-form comments suffer from the problem of decay, as they are difficult to maintain consistent with the source code. Additional research is required to determine whether any of these approaches present a more favorable trade-off between cost and usefulness than the current version of the concern graph approach.

6.4 The Importance of a Good Seed

Throughout this dissertation, in the description of the concern graph approach, we have assumed that a developer knows a relevant program location from where investigation can start. Based on such a starting point, or seed, a developer can investigate related elements in the source code and build a concern graph. Because concern graphs are designed to support a very focused investigation of the source code, the approach is not intended to assist with the broad type of investigation related to the identification of a seed. As described in Section 3.2.1, the identification of a seed is a separate phase of a program maintenance task, performed outside of the FEAT tool. There exists a variety of ways a developer can obtain a seed for the investigation of the code pertaining to a maintenance task. One can rely on other developers. This is the approach was have used in the AVID study (Section 4.2), and have simulated in the jEdit study (Section 4.5). Other possibilities include broad lexical searches for relevant keywords in all the source files, and specialized feature detection techniques (described in section 7.1.3).

During the evaluation of our technique for automatically inferring concern code from program investigation activities (Chapter 5), we observed that developers unfamiliar with a code base performed much more focused and effective program investigation if a good seed had been provided. This observation has two important consequences for potential adopters of the concern graph approach. First, one should only attempt to build a concern graph once a relevant seed has been identified; failing to do so may result in effort wasted documenting irrelevant information. Second, and more importantly, a database of concern graphs can provide an alternative source of potential seeds for other program evolution tasks. By perusing the concerns other developers have built for tasks similar to a task at hand, a developer can potentially discover a good seed. Section 6.6 discusses in more detail the improvements concern graphs can provide to the maintenance process.

6.5 Concern Interaction Analysis

One of the characteristics of the general concern graph model is the support for analyzing the interactions between two concerns. Given two concern definitions, concern interaction analysis produces a list of common participants between the concerns, and a list of relations between the participants of one concern and the participant of the other concern (see Section 2.3.1). We implemented support for concern interaction analysis in the FEAT tool in the form of the compare feature (see Section 3.2.2). In devising and implementing support for concern interaction analysis, our goal was to increase the usefulness of concern graphs by providing a means for developers to analyze whether and how two concerns interact without having to peruse the entire concern descriptions and perform the analysis mentally. We evaluated the contribution of the concern interaction analysis to the usefulness of the concern graph approach as part of the jEdit study. In this study, the training documentation for users of FEAT included detailed information about how to use concern interaction analysis, examples for subjects to practice using the feature, and instructions detailing the situations when concern interaction analysis could be useful. In spite of these provisions, none of the six subjects who performed the evolution task on jEdit with FEAT used concern interaction analysis in more than a cursory and exploratory way. Based on interview data, we established that the subjects had not used interaction analysis because it had not been deemed useful. Specifically, having just built a concern graph, the information captured by the concern graph was still fresh in the subjects' memory, and the concern interaction analysis was not seen as providing significant help for the task. The usefulness of the concern interaction analysis thus remains an open question. Our hypothesis is that, although it does not seem to be useful for developers initially investigating a concern, it might help other developers accessing the concern at a later stage. Further research should help us evaluate the usefulness of concern interaction analysis in different contexts.

6.6 The Influence of Concern Graphs on the Evolution Process

Having described how concern graphs can help developers perform software evolution tasks, we can now comment on the influence of the use of concern graphs on the software evolution process. As mentioned in the introduction, the process of modifying a software system can be separated into three phases: understanding the existing software, modifying the existing software, and re-validating the modified software. Using concern graphs to help in the evolution of a system does not change this fundamental decomposition, nor does it add additional steps to the process. However, more emphasis in put on the first phase, in the hope of achieving considerable benefits in the second.

Traditionally, during software evolution activities, more emphasis is put on the coding phase, to the detriment of program investigation. Nevertheless, it is our belief that a complete and thorough investigation of the implementation concerns involved in a change task significantly and positively influences the quality of a change. Indeed, the risks of performing software modification without fully understanding the implications of the change are well known [101]. Why, then, do developers immediately engage in source code modifications following a simple and desultory investigation? A potential justification for this practice it the perception, both by individuals and organizations, that some of the time spent investigating source code is wasted, while time spent coding translates into direct progress. Of course, nothing is further from the truth, as sloppy or incorrect program changes can often lead to disastrous consequences. Nevertheless, as long as the value of detailed analysis is not clearly and unequivocally demonstrated, the temptation will always exist to begin a software change with a limited understanding of the code. Our goal, with concern graphs, is to reduce the cost of program investigation by supporting a more systematic and focused process, and at the same time provide more value out of program investigation by supporting the creation of concern descriptions that can be used to support the software modification phase. It is our hope that, by providing a means of lessening the cost and augmenting the value of the initial analysis phase of program evolution, more developers will realize the importance of this activity.

Finally, by describing code relevant to a change, concern graphs have the potential to provide support in the third phase of software evolution: revalidation of the modified software. The investigation of this possibility is outside the scope of this dissertation. However, our work on concern graphs has already attracted the attention of researchers working on software testing [129].

6.7 Future Work

In the previous sections we have discussed different issues our past research on concern graphs has raised, and possible ways additional investigation can help us further our understanding of the impact of the concern graph approach, and of the way we can improve its effectiveness. Additionally, the work described in this dissertation has stimulated original research directions. In this section, we briefly discuss four new areas for future research involving concern graphs: automation of concern graph construction, research on concern databases, experimentation with pattern-based code investigation, and concern graph-based code refactoring.

6.7.1 Automatic Concern Graph Construction

From the onset, the idea of automatically creating concern graphs has shown promise. In Chapter 5, we presented a preliminary investigation of a technique for automatically creating concerns from program investigation activities. This technique can be customized for different styles of program investigation using a series of parameters. We envision the complete integration of the technique into a development environment that would allow users to choose between different parameter 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 the FEAT tool. 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.

We are also investigating an approach for automatic concern graph creation based on an analysis of the differences between two versions of the code base. Using this technique, a developer can take a snapshot of the program model of a project at the beginning of a change task, perform modifications to the source code, and then produce a concern graph representing the elements and relations present in the last version of the code that were not present in the first version. This technique has been implemented and integrated in the FEAT tool by a co-op student in the Software Practices Laboratory at the University of British Columbia. Compared to the technique described above, which requires monitoring program investigation activities, the code differencing technique can be applied to produce concerns using any version of the code available in a system's revision history. The concerns produced by this lasts technique, however, are typically less complete and descriptive than the ones created based on human input. An additional possibility for further research in this area is to experiment with a combination of the two approaches to automatic concern construction.

6.7.2 Concern Databases

One of the underlying goals of the concern graph representation is to allow organizations to accumulate, through repeated evolution tasks, a collections of concern descriptions for a system. Effective use of such databases will require tools and techniques to help developers find concern graphs of potential interest. We have collected a series of concern graphs during the development of the FEAT tool. Once a sufficient number is available, it will be possible to begin the proper investigation of the problems surrounding the querying of concern databases. This investigation should, in turn, allow us to evaluate how useful developers find concern graphs created by other developers for a different task.

6.7.3 Pattern-based Code Investigation

A concern graph represents a network of interactions between concrete elements defined in a program. By treating one or more of the participants in a concern graph as a variable, we obtain a template of interactions. For example, let us assume that a concern graph c captures the interactions `m1() calls m2()` and `m2() accesses f1`. The concern graph c thus captures two interactions between three program elements: two methods and one field. If we consider method `m2` to be a variable x instead of a concrete program element, we obtain the template concern $c(x)$, where x can take the value of any method in the program that is both called by `m1` and that accesses `f1`. Such templates can then be used to perform multi-predicate searches in a code base, to provide additional help during program investigation. We are planning to investigate possible mechanisms for specifying templates based on concern graphs, to investigate how to use these templates as the basis for searches, and to evaluate the usefulness of the results produced in the context of prgram evolution tasks.

6.7.4 Concern Graph-based Code Refactoring

In certain cases, scattered concerns can prove to be a constant burden on developers during repeated program evolution tasks. In such cases, it might be warranted to refactor the code base to explicitly modularize the offending concerns. Refactoring usually involves the modification of source code through a series of stereotypical changes that can be semi-automated [43]. For example, one typical refactoring consists in moving a method definition to its super-class. The Eclipse platform provides automated support for simple refactorings. One interesting use of concern graph is to specify source code that should be the target of such refactoring, and to provide support for automating the refactoring process. To account for scattered concerns, source code can be refactored either by modifying the system in its native programming language, but also by porting the system to a language supporting an advance separation of concerns mechanism (see Section 1.1). In both cases, a concern graph can form a basis for the semi-automatic refactoring process. We are currently investigating how concern graphs can provide support for refactoring concerns into aspects in the AspectJ language [68].

Chapter 7

Related Work

In this chapter, we discuss work related to the concern graph approach. We categorize related work into three groups: approaches proposed to help developers find source code relevant to a concern or change task (Section 7.1), approaches aimed at documenting and analyzing concerns in source code (Section 7.2), and approaches aimed at detecting and managing inconsistencies in software engineering artifacts (Section 7.3).

7.1 Concern Code Location

Many program understanding and reverse engineering tools and techniques have been proposed to help a developer discover the code related to a program evolution task. The different code location techniques described in the literature rely on a variety of information, such as the static structure of programs, program execution traces, and software documentation. In this section, we present an overview of the main types of code location techniques: use of cross-referencing tools, program slicing, feature location approaches, and code clustering techniques. Although all these techniques can help with the concern location problem, none of them supports the documentation of concerns.

7.1.1 Cross-referencing Tools

Cross-referencing tools, such as code browsers and program databases, allow developers to perform queries that elicit the relations between different program elements that may potentially be scattered in source code. The main purpose of cross-referencing tools is to provide developers with information that cannot be obtained easily through source code inspection. For example, one typical query supported by cross-referencing tools is the determination of the callers of a function or method.

Cross-referencing support has been an important part of some programming language environments. As early as 1981, the Interlisp programming environment included Masterscope [136], an interactive program supporting cross-referencing queries that provided developers with information such as the callers of a function or the accessors of variables. Similar functionality was integrated in environments for other languages, such as Smalltalk [47], and Trellis/Owl [94]. These languages and environments benefited from an early and integrated support for cross-reference queries partly because a program database formed an intrinsic component of their architecture.

Support for cross-referencing in more mainstream languages usually takes the form of separate, stand-alone tools. The C Information Abstractor (CIA) [25] is a program database supporting cross-reference queries for C programs [66]. The information used by CIA is collected through an analysis of the original source files. Tools have also been developed for C++ [132] programs, which present additional challenges to developers trying to understand scattered concerns, such as inheritance and dynamic binding. For example, the XREF/XREFDB system of Lejter et al. [72] allows developers to perform queries on a database of relations for a C++ program through the interface of the Emacs text editor [130].

Tools have also been developed to view structural program information collected in program databases. Ciao [24] is a graph-based navigator created to help developers view the relations produced by CIA. Rigi [81, 131] shows the relationships between different program elements in a graph representation.

Finally, recent environments have been developed for the Java language that support cross-reference querying. For example, the Eclipse platform [93] discussed earlier in this dissertation supports cross-reference queries. The JQuery tool of Janzen and De Volder [63], developed as an Eclipse plug-in, allows a developer to form specialized browsers to navigate code, and to perform queries in these browsers while retaining navigation context.

Although cross-referencing tools allow developers to find information that is potentially useful in identifying the source code relevant to a change, the context for collecting the information is limited. Specifically, the tools mentioned above do not support accumulating arbitrary results of queries in a network of program elements. In the cases where some context-sensitivity is provided, such as the browsers of JQuery, it is only in the form of a sequential history of queries; choosing a non-contiguous subset is not possible. As a result, when using these tools, a developer must manually build a list of program elements pertaining to a concern and manage the context in which these elements are used and queried. In brief, although cross-reference tools can help address the concern location problem, their support for program evolution tasks involving scattered concerns is limited by their lack of support for concern documentation.

7.1.2 Program Slicing

Program slicing denotes a type of analysis intended to identify the parts of a program that may affect the values computed at some point of interest [138]. Slicing was originally defined by Mark Weiser as a static analysis technique [146], but dynamic variants have since been developed. Slicing can be formulated as a graph reachability problem [98] on a program representation called the Program Dependence Graph (PDG) [40]. A PDG is a graph representing a combination of control- and data-flow dependences between statements in a program. Slicing can be used for many software engineering tasks such as parallelization, debugging [78, 145], or reverse engineering [10]. For maintenance activities, slicing can be used to help determine the impact of changes [45]. Visual techniques have also been developed to help in this process [44, 60].

Many variants of slicing have been proposed to deal with technical issues such as the slicing of programs with procedures [58], or of object-oriented programs [75, 139]. In particular, Jackson and Rollins have proposed *chopping* [62], a generalization of slicing based on a different program dependence graph supporting both a modular treatment of procedures, and a fine-grained slicing taking into account the influence of individual variables (as opposed to statements). Chopping has been shown to produce more accurate results than traditional slicing based on the PDG.

Although they are conceptually appealing techniques, static slicing and its variants suffer from many practical limitations. First, computing slices can be expensive [146]; pragmatic considerations may require lower-precision data-flow analyses [76, 140], resulting in coarser, more conservative results. Furthermore, trade-offs related to the analysis of programs with pointers lead to additional conservativeness [56]. Finally, because a statement is often transitively dependent on many other statements, slices are often very large [62, 146]. This problem is only exacerbated by more conservative analyses.

Dynamic slicing [1, 52] is another variant of slicing that takes into account program execution trace information. Specifically, dynamic analysis only considers program dependences that occur in a specific execution of the program. As such, the input to dynamic slicing tools must include a representation of the execution of a program based on a specific input.

For the purpose of helping developers find code relevant to a concern, one major limitation of slicing is that only one type of concern can be identified: code related through a control- and data-flow criterion. As such, slicing cannot be used to automatically infer the code relevant to a concern. For example, code relevant to a concern but unrelated in the program, such as code exchanging data through a file, may not be identified by slicing. Finding this code requires human intervention. Another drawback of slicing is that it does not discriminate between interesting and boilerplate code. For this reason also, slicing cannot be used to automatically infer concerns, since a slice is bounded by a graph reachability criterion, as opposed to a human-centric evaluation of relevance. In brief, the results produced by slicing will not always correspond to the code relevant to concerns a developer has when changing a program. Even if slicing can be used to help a developer reason about the impact of a statement on the behavior of a program, at some point, the developer will need to investigate source code manually (or semi-automatically), to focus on specific areas of interest. Concern graphs have been designed to support this latter activity.

7.1.3 Feature Location Techniques

Different techniques have been proposed to help developers identify parts of the source code that implement user-level features.

The Software Reconnaissance technique developed by Wilde et al. identifies features in source code based on a analysis of the execution of a program [150, 149]. 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. Given a feature f and test cases that exercise and do not exercise f, the analysis produces four sets of *components*:

- *Common components*: the set of components exercised in all test cases;
- *Components potentially involved with* f: the set of components that are executed in at least one test case that exhibits f;
- *Components indispensably involved in* f: the set of components exercised in all of the test cases exhibiting f;
- *Components uniquely involved in* f: the set of components that are exercised in some test case exhibiting f and excluding any component exercised in any test case that do not exhibit f.

The technique does not explicitly define the notion of component, and as such different definitions can be used (e.g., functions, statements). However, based on their experience, the inventors of the technique have found that the most useful definition of component is that of the control-flow arc [150]. Experience with the technique on industrial systems ranging between 10 and 28 kLOC of C and C++ code showed that, although the technique could not always find all or even some of the source code implementing a feature, the set of components uniquely involved in f usually provided a good starting point for developers to investigate the source code [148]. 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 [77].

Wong et al. proposed an approach that is also based on the analysis of the difference between program execution traces [151]. The approach of Wong et al. provides results similar to the Software Reconnaissance technique, but, in addition, presents results at different levels of granularity (e.g., files, lines of code, blocks).

A third approach to feature location based on dynamic analysis was developed by Eisenbarth et al. [38, 39]. Eisenbarth et al. produce the mapping between components and test cases using mathematical concept analysis (a partial ordering and clustering technique [126]). In addition to producing a basic mapping between components and test cases, the approach of Eisenbarth et al. involves the refinement of the feature-to-code mapping through inspection by a developer of a static dependency graph of the program analyzed. This step helps achieve a more precise and complete description of the code implementing a feature, at the cost of additional effort for developers using the technique.

Software Reconnaissance and the respective approaches proposed by Wong, Eisenbarth, and their colleagues, like any dynamic analysis approach, depend on the availability and quality of test cases for an executable system. In contrast, the construction of concern graphs is based on source code, and can be applied to incomplete or incorrect code. As these approaches based on dynamic analysis have achieved some success in identifying good starting points for program investigation, they can be considered complementary to the use of concern graphs to support program evolution tasks.

A semi-automatic technique for feature location based on static analysis has been proposed by Chen and Rajlich [23]. Using this technique, a developer navigates a system dependency graph computed through a static analysis of the source code of a program. The graph produced is a model of a program not unlike the model we use for concern graphs. It details relations between globally-defined elements (e.g., functions and global variables in a C program). The technique involves a systematic, computer-assisted search through the dependency graph to find elements related to a feature. This approach is limited in that it does not allow users to find concern elements that are related through a non-concern element, since the technique dictates that the search must stop on a path once a unrelated element is reached. In a recent study [147], Wilde et al. have compared the dependency graph method of Chen and Rajlich to the Software Reconnaissance method on legacy Fortran code. The results showed that, although both methods were successful in identifying code relevant to a feature, Software Reconnaissance was better suited to large and infrequently changed programs, and the dependency graph method was better suited to programs that require a deep investigation by developers.

Finally, Antoniol et al. have proposed an approach to determine a set of components potentially affected by a maintenance tasks using a probabilistic analysis of the text of the maintenance request [4]. This approach, however, produces results only at the granularity of high-level component (classes), and could not be used to produce concern graphs.

There are two important distinctions between the work discussed above and the concern graph approach. The first lies in the nature of the concerns analyzed. All the approaches above focus on identifying the code relevant to features that can be expressed at the user level. These form a proper subset of the concerns a developer might wish or need 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, the concern graph approach is flexible enough to capture any subset of a program as a concern. A second important difference is that the approaches discussed in this section focus primarily on finding the source code implementing a feature, whereas the primary goal of a concern graph is to document this information in a robust fashion.

7.1.4 Clustering Techniques

Some design recovery techniques have been proposed to identify code that would constitute a candidate for refactoring into a module or object. These approaches are typically based on the analysis of relations between different program elements, such as "x uses y", and determine cohesive program subsets using various clustering algorithms. For example, de Oca and Carver propose an approach to identify data cohesive subsystems in COBOL programs using data mining techniques [31]. van Deursen and Kuipers report on the use of both cluster and concept analysis to identify potential objects in non-object-oriented code [142]. Siff and Reps [123], and Tonella [141] both propose approaches to module identification in legacy systems based on 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 or implementation concept. In contrast, our approach to finding source code relevant to concerns, both with and without automation, factors in the focus of the developers during program investigation, allowing the capture of concerns that are of immediate interest to program developers.

7.2 Concern Documentation

Scattered concerns are a fundamental issue in software development, and many approaches have been proposed that involve the explicit description and documentation of concerns to aid in various software engineering tasks. Mechanisms for describing concerns have been proposed for tasks at different stages of the software development process (e.g., requirement specification [105], and design [30]). Descriptions of concerns also span the full spectrum of levels of abstraction, with some approaches supporting the definition of concern corresponding to features at the architectural level, and the generation of code to implement these concerns [9] in the context of software product lines [49]. Not all of these approach address the problem of finding and documenting concerns in source code. In this section, we discuss the approaches specifically addressing the problem of describing concerns at the implementation level.

7.2.1 Textual Documentation

Early empirical evidence that scattered concerns pose problems to programmers was collected by Soloway, Letovsky, et al. during different studies of professional programmers [74, 127]. In one study, conducted at NASA's Jet Propulsion Laboratory, Soloway et al. observed that the programmers who did not implement a correct modification to a small system "failed to understand the casual interactions inherent in one of the key delocalized plans." [127: p. 1262]. To address the difficulty of performing maintenance on code involving delocalized plans (or, in other words, scattered concerns), the researchers propose that programmers produce explicit documentation detailing delocalized plans in programs. Their initial approach is a form of paper documentation where source code is presented in parallel with pointers linking the code to other relevant sections of a program, and detailing the rationale for different design and implementation decisions. The authors also mention the possibility of computer-assisted documentation, but do not elaborate. Although the idea of Soloway et al. is based on sound empirical observations, their proposed solution has several limitations. First and foremost, no evaluation of its cost-effectiveness is performed, and we can surmise that the real cost of pre-emptively documenting scattered concerns is high. Furthermore, this cost may not always be warranted as some scattered concerns may never be revisited. Finally, textual documentation in plain language suffers from the problem of decay: in order to remain consistent, it must be updated in parallel with the code. This introduces the possibility of misleading discrepancies between the code and the documentation creeping in as a program goes through many modifications. Compounding this problem are the facts that human-produced documentation can be wrong, and that there is no way to automatically detect inconsistencies between plain-language documentation and source code. For these reasons, the documentation technique proposed by Soloway et al. is not practical. The concern graph approach proposed in this dissertation shares the goal of documenting scattered concerns, but addresses most of the limitations of a manual approach. In particular, using concern graphs, concern documentation, can be produced at a minimal cost, and inconsistencies between the documentation and source code can be automatically detected and repaired. The trade-off for these advantages is a lack of support for documenting design rationale in concern graphs that is possible in free-form documentation.

7.2.2 Conceptual Modules

A different approach to documenting scattered concerns is the idea of conceptual modules [7]. The intent of the conceptual modules approach is to allow a developer to query a program both in terms of the existing and of a desired structure. In practice, a conceptual module captures segments of a program as a list of lines of source code. The approach is supported by a tool that can produce information such as input, output, and local variables for a conceptual module, the definitions and uses of variables in a conceptual module, calls made to and by code in the module, and relationship information between conceptual modules. The goals of the concern graph and conceptual module approaches are different; the goal of conceptual modules is to allows precise queries on source code, whereas the goal of concern graphs is to capture knowledge about the implementation of a concern in source code. This divergence in point of view translates into a practical differences: conceptual modules do not abstract the essential structure of a concern. As a consequence, conceptual modules can only exist on one version of a system, and cannot be used to described knowledge about the implementation of a concern through a program's life-cycle.

7.2.3 Concern Visualization Tools

Concerns can also be described in terms of subsets of the program text matching different queries. The Aspect Browser is a tool developed to help developers find concerns using lexical searches of the program text [50]. Concerns found in this fashion can be stored and viewed at different times to support program evolution tasks. Aspect Browser uses the Seesoft [37] concept and a map metaphor to graphically represent the location of code implementing concerns in the context of the entire code base. The Aspect Mining Tool (AMT) [54] is conceptually similar to Aspect Browser, but supports additional queries based on types. The Aspect Browser and AMT can be used both for finding and documenting concerns. However, because they only support the specification of concerns based on lexical matches to regular expressions and use of types, their expressive power is limited. Additionally, these tools do not support the detection and repair of inconsistencies between a concern and a code base. Finally, the text-oriented approach also limits the tools' ability to capture relationships between scattered program elements explicitly.

7.2.4 Virtual Files

Descriptions of scattered concerns can also be captured as *virtual files*. In software development environments, the idea of virtual files is to present various segments of source code and other system documentation relevant to a task as a single unit. For example, the Desert Environment [106] explicitly supports the concept of virtual files. In Desert, a developer can load a virtual file consisting of fragments of other source files, and edit the virtual file. The system provides the logic for saving the fragments after they are edited. The system also provides support to build fragment files from a list of fragment names. The Stellation system [28, 29] is a fine-grained software configuration management system that supports method-level storage management. Using a concept similar to Desert, Stellation is intended to supports the concept of virtual source files using a typed aggregation mechanism that supports the collection of different program elements and other artifacts (such as test cases) in a single unit for the purpose of configuration management. Besides explicit specification, the proposal for Stellation includes the possibility of specifying aggregates in terms of query results.

Virtual files can provide a means of documenting scattered code that implements a concern. However, these mechanisms do not address the concern location problem, and virtual files must be composed by a developer who already knows about the location of the code implementing a concern. The mechanisms proposed for Desert and Stellation also do not include support for tolerating and managing inconsistencies between a virtual file and the source code. In particular, the proposal for Stellation does not detail how externalized virtual files relying on queries can be re-generated in the presence of inconsistencies.

7.2.5 Advanced Separation of Concerns Mechanisms

Finally, concerns can be captured explicitly by changing the source code to factor the code relevant to a concern into a special module. This functionality is supported by advanced separation of concerns mechanisms. Section 1.1 presents an overview of such mechanisms, and of the issues they address. Changing the source to explicitly modularize scattered concerns is a very different approach than the use of concern graphs. First, the cost and associated risks of re-modularizing a program in an separate language are much higher, and as such the change might not be warranted in all cases. Second, advanced separations of concerns mechanism are not flexible enough to support the encapsulation of all the potential concerns emerging during program evolution. In contrast, concern graphs provide an inexpensive way to describe concerns in programs without requiring any change to the code.

7.3 Inconsistency Management

There exists a large body of work in computer science addressing the problem of managing the consistency between different pieces of information. In this section, we provide an overview of the significant work in the area of software engineering, and discuss the relevance of our work on concern graphs.

One expression of the need for inconsistency management came with the apparition of software development environments (e.g., Centaur [19], Arcadia [65, 135]). Such tools require the management of consistency between different artifacts related to a program (e.g., textual views of the source code, abstract syntax trees, and control-flow graphs). To help address this problem, frameworks have been proposed that explicitly account for consistency management [53, 86, 133].

An important development in the research on consistency management was the realization that enforcing total consistency might not always be possible or even desirable [41]. This idea was originally proposed by Balzer [6] based on research on data management. Balzer proposes to temporarily tolerate certain inconsistencies, by marking inconsistent data using “pollution markers”. We have retained this approach to implement inconsistency management for concern graphs.

Research on inconsistency management has also been done to support requirement specifications [55, 92]. In the area of design, Finkelstein et al. studied the issues related to inconsistency handling in situations where potentially overlapping elements of design (design fragments) can be specified by different developers having different perspectives [42]. However, work on inconsistency checking with viewpoints has remained of a mostly theoretical nature [90].

Lastly, with the advent of development systems distributed over the Internet, new problems have appeared related to the management of the consistency of distributed, heterogeneous data. Systems have been proposed to address this new problem. For example, xlinkit, a lightweight framework for consistency checking [87, 88, 89, 90], supports the incremental detection and repair of inconsistencies in a web of heterogeneous, distributed software engineering documents.

Most of the published research on consistency management has focused on developing frameworks supporting a holistic and unified view of inconsistency management. This strategy has the advantage of providing sound solutions that apply in a variety of situations involving a variety of data. For example, the rules supported by the xlinkit environment mentioned above support checks to the documents both within and across development stages. An important tradeoff of this general approach is that additional effort must be spent implementing the framework for a desired situation: consistency rules have to be written and tested, and developers have to be trained to use them. Additionally, by supporting a solution to a general problem, frameworks cannot leverage from the semantics of specific inconsistencies to provide additional help to developers.

In our effort to minimize the cost of using concern graphs, we have instead developed a specialized approach to consistency management. The mechanism we developed for concern graphs has a simple model that can detect a single type of inconsistency: mismatches between a fragment projection and a model of the source code. Our mechanism also involves a small and fixed number of consistency rules that do not need to be updated by users of concern graphs. Indeed, by defining our notion of inconsistency at the level of the general program model for concern graphs, we ensure that management of inconsistencies is independent of any concrete program model instantiated. Finally, and perhaps most importantly, having a dedicated inconsistency management mechanism allowed us to provide tool support for managing specific consistency rules. As we found (see Section 4.6), dedicated tool support for inconsistency management is a powerful feature that supports not only detecting and repairing inconsistencies, but also reasoning about their cause with the help of visualization and queries. In brief, the contribution of our approach in the area of inconsistency management is the demonstration that a specialized mechanism can provide support for reasoning about inconsistencies that goes beyond simple detection and repair. For example, as we have shown in Section 4.6, the support we have implemented for detecting and visually representing inconsistent fragments in FEAT allowed us to determine the indirect cause of an inconsistency, and to manually repair the inconsistency to account for this cause.

Chapter 8

Conclusions

Evolving programs can be a difficult task, especially when it requires a developer to locate and understand scattered concerns—considerations a developer might have about the implementation of a program which are not implemented in a single location in the code.

The motivation for the work described in this dissertation is to help developers locate and understand scattered concerns, and to document the code relevant to these concerns so that knowledge about their implementation needs not be repeatedly re-acquired. To achieve this goal, we propose, during program investigation activities, to produce descriptions of the code implementing a concern. Capturing concern representations can thus support both the investigation at hand, and later tasks involving the same concerns.

As such, the thesis of this dissertation has been that a description of concerns, representing program structures and linked to source, that can be produced cost-effectively during program investigation activities, can help developers perform software evolution tasks more systematically, and on different versions of a system.

To investigate the claims expressed in this thesis statement, we developed the concept of concern graphs, a model for describing concerns in source code based on relations between elements defined in a program. The concern graph model is general, and can be instantiated to capture different types of relations between different types of elements in different programming languages. The model also defines precisely the notion of inconsistency between a concern graph and the corresponding source code. To experiment with concern graphs, we have developed a tool, called FEAT, that allows developers to iteratively build concern descriptions as the source is investigated, to view the code related to a concern, and to perform analyses on the concern representation. We have also developed an algorithm to automatically generate concern graphs based on a transcript of program investigation activities. Using FEAT, we have evaluated the cost and usefulness of concern graphs in a series of case studies involving the evolution of five different systems of different size and style. The results show that concern graphs are inexpensive to create during program investigation, can help developers perform program evolution tasks more systematically, and are robust enough to be used with different versions of a system.

In addition to demonstrating the validity of the thesis statement, the research described in this dissertation makes six contributions to the field of software engineering.

First, we provide a general model for describing concerns in source code. As a consequence of the generality of the model, the analyses and tool support required to support concern graphs in different languages can, to a large extent, be reused.

Second, we provide a specific instantiation of the model for the Java language, and a discussion of the issues of usability and scalability related to the use of this specific model. Other researchers can rely on this knowledge to extend or adapt our model to suit different purposes.

Third, we provide a usable tool capable of supporting the concern graph approach for Java programs. Researchers and developers can download our tool freely to conduct research on separations of concerns and modularity, and to integrate the use of concern graphs in industrial settings.

Fourth, we describe an algorithm that can automatically infer concerns from a transcript of the program investigation of a developer. This algorithm serves as a proof of concept that such a technique is possible, and that it can produce documentation for scattered concerns at a minimal cost.

Fifth, we provide an in-depth description of the design of five empirical studies of program evolution, of the issues and problems we have encountered, and of the steps we have taken to address them. This knowledge can be useful to researchers wishing to develop similar studies of programmers performing software evolution tasks.

Finally, we demonstrate a specialized mechanism for the management of inconsistencies between a description of source code and an actual code base that can provide support for reasoning about the indirect cause of an inconsistency, in addition to the simple detection and repair of inconsistencies.

In conclusion, although our approach is still at an early stage, the idea of using concern graphs to support program evolution tasks shows promise, and we envision its eventual deployment in an industrial setting.

Bibliography

[1] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, pages 246–256. ACM Press, New York, NY, USA, June 1990. 97

[2] Alfred V. Aho. Pattern matching in strings. In Ronald V. Book, editor, *Formal Language Theory: Perspectives and Open Problems*, pages 325–347. Academic Press, New York, NY, USA, 1980. 10

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*, chapter 6: Type Checking. Addison-Wesley Publishing Company, Reading, MA, USA, 1986. 18

[4] G. Antoniol, G. Canfora, A. De Lucia, and E. Merlo. Recovering code to documentation links in OO systems. In *Proceedings of the 6th Working Conference on Reverse Engineering*, pages 136–144. IEEE Computer Society Press, Los Alamitos, CA, USA, October 1999. 99

[5] David F. Bacon. Fast and effective optimization of statically typed object-oriented languages. Ph.D. Thesis CSD-98-1017, University of California, Berkeley, CA, USA, October 1998. 43, 122

[6] Robert Balzer. Tolerating inconsistency. In *Proceedings of the 13th International Conference on Software Engineering*, pages 158–165. IEEE Computer Society Press, Los Alamitos, CA, USA, May 1991. 42, 102

[7] Elisa L.A. Baniassad and Gail C. Murphy. Conceptual module querying for software reengineering. In *Proceedings of the 20th International Conference on Software Engineering*, pages 64–73. IEEE Computer Society, Los Alamitos, CA, USA, April 1998. 10, 11, 100

[8] Elisa L.A. Baniassad, Gail C. Murphy, Christa Schwanninger, and Michael Kircher. Managing crosscutting concerns during software evolution tasks: An inquisitive study. In *Proceedings of the 1st Conference on Aspect-Oriented Software Development*, pages 120–126. ACM Press, New York, NY, USA, April 2002. 9, 10, 32

[9] Don Batory, Roberto E. Lopez-Herrejon, and Jean-Philippe Martin. Generating product-lines of product-families. In *Proceedings of the 17th International Conference on Automated Software Engineering*, pages 81–92. IEEE Computer Society Press, Los Alamitos, CA, USA, September 2002. 99

[10] Jon Beck and David Eichmann. Program and interface slicing for reverse engineering. In *Proceedings of the 15th International Conference on Software Engineering*, pages 509–518. IEEE Computer Society Press, Los Alamitos, CA, USA, May 1993. 96

[11] Kent Beck. Embracing change with extreme programming. *IEEE Computer*, 32(10):70–77, October 1999. 5

[12] Laszio A. Belady and M. M. Lehman. A model of large program development. *IBM Systems Journal*, 15(3):225–252, 1976. 1

[13] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas E. Webster. Program understanding and the concept assignment problem. *Communications of the ACM*, 37(5):72–82, May 1994. 1

[14] Barry W. Boehm. Software engineering. *IEEE Transactions on Computers*, 12(25):1226–1242, December 1976. 1

[15] B.W. Boehm, J.R. Brown, and M. Lipow. Quantitative evaluation of software quality. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 592–605. IEEE Computer Society Press, Los Alamitos, CA, USA, October 1976. 1

[16] Shawn A. Bohner. Software change impact analysis for design evolution. In *Proceedings of the 8th International Conference on Software Maintenance and Re-engineering*, pages 292–301. IEEE Computer Society Press, Los Alamitos, CA, USA, 1991. 1

[17] Shawn A. Bohner and Robert S. Arnold. *An Introduction to Software Change Impact Analysis*, pages 1–26. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996. 1

[18] Shawn A. Bohner and Robert S. Arnold, editors. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996. 5

[19] P. Borras, D. Clement, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the system. In *Proceedings of the 3rd ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 14–24. ACM Press, New York, NY, USA, November 1988. 102

[20] Lars Bratthall and Magne Jørgensen. Can you trust a single data source exploratory software engineering case study? *Empirical Software Engineering*, 7(1):9–26, March 2002. 56

[21] Tim Bray, Jean Paoli, C.M. Sperberg-McQueen, and Eve Maler, editors. *Extensible Markup Language (XML) 1.0*. W3C, 2nd edition, 2000. 42

[22] Per Cederqvist. *Version Management with CVS*. Signum Support AB, Linköping, Sweden, November 1993. 10

[23] Kunrong Chen and Václav Rajlich. Case study of feature location using dependence graph. In *Proceedings of the 8th International Workshop on Program Comprehension*, pages 241–247. IEEE Computer Society Press, Los Alamitos, CA, USA, 2000. 98

[24] Yih-Farn Chen, Glenn S. Fowler, Eleftherios Koutsofios, and Ryan S. Wallach. CIAO: A graphical navigator for software and document repositories. In *Proceedings of the International Conference on Software Maintenance*, pages 66–75. IEEE Computer Society Press, Los Alamitos, CA, USA, 1995. 96

[25] Yih-Farn Chen, Michael Y. Nishimoto, and C.V. Ramamoorthy. The C information abstraction system. *IEEE Transactions on Software Engineering*, 16(3):325–334, March 1990. 10, 30, 95

[26] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, January 1990. 10

[27] Jong-Deok Choi, David Grove, Michael Hind, and Vivek Sarkar. Efficient and precise modeling of exceptions for the analysis of Java programs. In *Proceedings of the ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering*, pages 21–31. ACM Press, New York, NY, USA, September 1999. 18, 31

[28] Mark Chu-Caroll, James Wright, and David Shields. Supporting aggregation in fine grained software configuration management. In *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 99–108. ACM Press, New York, NY, USA, November 2002. 101

[29] Mark C. Chu-Carroll and Sara Spenkle. Coven: Brewing better collaboration through software configuration management. In *Proceedings of the ACM SIGSOFT 8th International Symposium on the Foundations of Software Engineering*, pages 88–97. ACM Press, New York, NY, USA, November 2000. 101

[30] Siobhán Clarke, William Harrison, Harold Ossher, and Peri Tarr. Subject-oriented design: Towards improved alignment of requirements, design and code. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 325–339. ACM Press, New York, NY, USA, November 1999. 99

[31] Carlos Montes de Oca and Doris L. Carver. Identification of data cohesive subsystems using data mining techniques. In *Proceedings of the International Conference on Software Maintenance*, pages 16–23. IEEE Society Press, Los Alamitos, CA, USA, November 1998. 99

[32] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the European Conference on Object-oriented Programming*, volume 952 of *Lecture Notes in Computer Science*, pages 77–101. Springer-Verlag, Heidelberg, Germany, August 1995. 43, 122

[33] Edsger W. Dijkstra. The structure of the ”THE”-multiprogramming system. *Communications of the ACM*, 11(5):341–346, May 1972. 3

[34] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, USA, 1976. 3

[35] Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, February 1970. 18

[36] Stephen G. Eick, Todd L. Graves, Alan F. Karr, J.S. Marron, and Audis Mockus. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, 2001. 5

[37] Stephen G. Eick, Joseph L. Steffen, and Eric E. Summer, Jr. Seesoft—A tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, November 1992. 3, 101

[38] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Derivation of feature component maps by means of concept analysis. In *Proceedings of the 5th European Conference on Software Maintenance and Reengineering*, pages 176–179. IEEE Computer Society Press, Los Alamitos, CA, USA, March 2001. 98

[39] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, 29(3):210–224, March 2003. 98

[40] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987. 15, 29, 96

[41] Anthony Finkelstein. A foolish consistency: Technical challenges in consistency management. In *Proceedings of the 11th International Conference on Database and Expert Systems Applications*, volume 1873 of *Lecture Notes in Computer Science*, pages 1–5. Springer-Verlag, Heidelberg, Germany, September 2000. 102

[42] Anthony C. W. Finkelstein, Dov Gabbay, Anthony Hunter, Jeff Kramer, and Bashar Nuseibeh. Inconsistency handling in multiperspective specifications. *IEEE Transactions on Software Engineering*, 20(8):569–578, August 1994. 102

[43] Martin Fowler. *Refactoring—Improving the Design of Existing Code*. Object Technologies Series. Addison-Wesley, Boston, MA, USA, 2000. With contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts. 5, 87, 94

[44] Keith B. Gallagher. Visual impact analysis. In *Proceedings of the International Conference on Software Maintenance*, pages 52–58. IEEE Computer Society Press, Los Alamitos, CA, USA, November 1996. 96

[45] Keith Brian Gallagher and James R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, August 1991. 10, 96

[46] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns—Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley Longman, Inc., Reading, MA, USA, 1995. 4, 5

[47] Adele Goldberg. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley Publishing Company, Reading, MA, USA, 1984. 10, 95

[48] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley Longman, Inc., Reading, MA, USA, 2nd edition, 2000. 3, 121

[49] Martin L. Griss. Implementing product-line features with component reuse. In *Proceedings of the 6th International Conference on Software Reuse*, volume 1844 of *Lecture Notes in Computer Science*, pages 137–152. Springer-Verlag, Heidelberg, Germany, June 2000. 99

[50] William G. Griswold, Jimmy J. Yuan, and Yoshikiyo Kato. Exploiting the map metaphor in a tool for software evolution. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 265–274. IEEE Computer Society Press, Los Alamitos, CA, USA, May 2001. 11, 101

[51] Object Management Group. *OMG Unified Modeling Language Specification*, 2000. Version 1.3. 30

[52] Tibor Gyimóthy, Árpád Beszédes, and Istán Forgács. An efficient relevant slicing method for debugging. In *Proceedings of the 7th European Software Engineering Conference and 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, volume 1687 of *Lecture Notes in Computer Science*, pages 303–321. Springer-Verlag, Heidelberg, Germany, September 1999. 97

[53] Torben Mejlvang Hagensen and Bent Bruun Kristensen. Consistency in software system development: Framework, model, techniques & tools. In *Proceedings of the 5th ACM SIGSOFT Symposium on Software Development Environments*, pages 58–67. ACM Press, New York, NY, USA, December 1992. 102

[54] Jan Hannemann and Gregor Kiczales. Overcoming the prevalent decomposition in legacy code. Position paper for the ICSE 2001 Workshop on Advanced Separation of Concerns in Software Engineering, May 2001. 101

[55] Constance L. Heitmeyer, Ralph D. Jeffords, and Bruce G. Labaw. Automated consistency checking of requirement specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, July 1996. 102

[56] Michael Hind and Anthony Pioli. Which pointer analysis should i use? In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 113–123. ACM Press, New York, NY, USA, August 2000. 97

[57] Susan Horwitz and Thomas Reps. The use of program dependence graphs in software engineering. In *Proceedings of the 14th International Conference on Software Engineering*, pages 392–411. ACM Press, New York, NY, USA, May 1992. 15

[58] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990. 96

[59] James W. Hunt and Thomas G. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5):350–353, May 1977. 69

[60] Matthew Hutchins and Keith Gallagher. Improving visual impact analysis. In *Proceedings of the International Conference on Software Maintenance*, pages 294–303. IEEE Computer Society Press, Los Alamitos, CA, USA, November 1998. 96

[61] D.C. Ince. *An Introduction to Discrete Mathematics, Formal System Specification, and Z.* Oxford Applied Mathematics and Computing Science Series. Clarendon Press, Oxford, 2nd edition, 1992. 119

[62] Daniel Jackson and Eugene J. Rollins. A new model of program dependence for reverse engineering. In *Proceedings of the 2rd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 2–10. ACM Press, New York, NY, USA, December 1994. 29, 96, 97

[63] Doug Janzen and Kris De Volder. Navigating and querying code without getting lost. In *Proceedings of the Conference on Aspect-Oriented Software Development*. ACM Press, New York, NY, USA, March 2003. 96

[64] Magne Jørgensen, Dag I.K. Sjøberg, and Geir Kirkebøen. The prediction ability of experienced software maintainers. In *Proceedings of the 4th European Conference on Software Maintenance and Reengineering*, pages 93–99. IEEE Computer Society Press, Los Alamitos, CA, USA, February 2000. 1

[65] R. Kadia. Issues encountered in building a flexible software development environment. In *Proceedings of the 5th ACM SIGSOFT Symposium of Software Development Environments*, pages 169–180. ACM Press, New York, NY, USA, December 1992. 102

[66] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, NJ, 2nd edition, 1988. 3, 96

[67] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):51–57, October 2001. 4

[68] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer-Verlag, Heidelberg, Germany, June 2001. 4, 94

[69] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, Heidelberg, Germany, June 1997. 1, 3, 87

[70] Barbara A. Kitchenham, Shari Lawrence Pfleeger, Lesley M. Pickard, Peter W. Jones, David C. Hoaglin, Khaled El Emam, and Jarett Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, 28(8):721–734, August 2002. 48

[71] M.M. Lehman and L.A. Belady. *Program Evolution: Processes of Software Change*, volume 27 of *APIC Studies in Data Processing*. Academic Press, Inc., London, UK, 1985. 5

[72] Moises Lejter, Scott Meyers, and Steven P. Reiss. Support for maintaining object-oriented programs. *IEEE Transactions on Software Engineering*, 18(12):1045–1052, December 1992. 96

[73] Stanley Letovsky and Elliot Soloway. Strategies for documenting delocalized plans. In *Proceedings of the Conference on Software Maintenance*, pages 144–151. IEEE Computer Society Press, Los Alamitos, CA, USA, November 1985. 2

[74] Stanley Letovsky and Elliot Soloway. Delocalized plans and program comprehension. *IEEE Software*, 3(3):41–49, May 1986. 1, 100

[75] Donglin Liang and Mary Jean Harrold. Slicing objects using system dependence graphs. In *Proceedings of the International Conference on Software Maintenance*, pages 358–367. IEEE Computer Society Press, Los Alamitos, CA, USA, November 1998. 96

[76] Donglin Liang and Mary Jean Harrold. Efficient points-to analysis for whole-program analysis. In *Proceedings of the 7th European Software Engineering Conference and 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 1687 of *Lecture Notes in Computer Science*, pages 199–215. Springer-Verlag, Heidelberg, Germany, September 1999. 97

[77] Kazimiras Lukoit, Norman Wilde, Scott Stowell, and Tim Hennessey. TraceGraph: Immediate visual location of software features. In *Proceedings of the International Conference on Software Maintenance*, pages 33–39. IEEE Computer Society Press, Los Alamitos, CA, USA, October 2000. 98

[78] James R. Lyle and Mark Weiser. Automatic program bug location by program slicing. In *Proceedings of the 2nd International Conference on Computers and Applications*, pages 877–882. IEEE Computer Society Press, Los Alamitos, CA, USA, 1987. 96

[79] James Martin and Carma McClure. *Software Maintenance: The Problem and Its Solutions*. Prentice-Hall, 1983. 1

[80] Robert Moreton. A process model for software maintenance. *Journal of Information Technology*, 5:100–104, 1990. 1

[81] Hausi A. Müller and Karl Klashinsky. Rigi—A system for programming-in-the-large. In *Proceedings of the 10th International Conference on Software Engineering*, pages 80–86. IEEE Computer Society Press, Los Alamitos, CA, USA, April 1988. 10, 96

[82] Gail C. Murphy, William G. Griswold, Martin P. Robillard, Jan Hannemann, and Wesley Leong. Design recommendations for concern elaboration tools. In Tzilla Elrad, Siobhán Clarke, Mehmet Aksit, and Robert Filman, editors, *Aspect-oriented Software Development*. Addison-Wesley Longman, Inc., Reading, MA, USA, 2004. To appear. 88

[83] Gail C. Murphy, Albert Lai, Robert J. Walker, and Martin P. Robillard. Separating features in source code: An exploratory study. In *Proceedings of the 23rn International Conference on Software Engineering*, pages 275–284. IEEE Computer Society Press, Los Alamitos, CA, USA, May 2001. 5

[84] Gail C. Murphy, David Notkin, William G. Griswold, and Erica S. Lan. An empirical study of static call graph extractors. *ACM Transactions on Software Engineering and Methodology*, 7(2):158–191, April 1998. 18

[85] Gail C. Murphy, Robert J. Walker, Elisa L.A. Baniassad, Martin P. Robillard, Albert Lai, and Mik A. Kersten. Does aspect-oriented programming work? *Communications of the ACM*, 44(10):75–77, October 2001. 5

[86] K. Narayanaswamy and Neil Goldman. ”lazy” consistency: a basis for cooperative software development. In *Proceedings of the 1992 ACM Conference on Computer-supported cooperative work*, pages 257–264. ACM Press, New York, NY, USA, November 1992. 102

[87] Christian Nentwich, Licia Capra, Wolfgang Emmerich, and Anthony Finkelstein. xlinkit: a consistency checking and smart link generation service. *ACM Transactions on Internet Technology*, 2(2):151–185, May 2002. 102

[88] Christian Nentwich, Wolfgang Emmerich, and Anthony Finkelstein. Static consistency checking for distributed specifications. In *Proceedings of the 16th International Conference on Automated Software Engineering*, pages 115–124. IEEE Computer Society Press, Los Alamitos, CA, USA, November 2001. 102

[89] Christian Nentwich, Wolfgang Emmerich, and Anthony Finkelstein. Consistency management with repair actions. In *Proceedings of the 25th International Conference on Software Engineering*, pages 455–464. IEEE Computer Society Press, Los Alamitos, CA, USA, May 2003. 102

[90] Christian Nentwich, Wolfgang Emmerich, Anthony Finkelstein, and Ernst Ellmer. Flexible consistency checking. *ACM Transactions on Software Engineering and Methodology*, 12(1):28–63, January 2003. 102

[91] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999. 18

[92] Bashar Nuseibeh, Steve Easterbrook, and Alessandra Russo. Leveraging inconsistency in software development. *IEEE Computer*, 33(4):24–29, April 2000. 102

[93] Object Technology International, Inc. Eclipse platform technical overview. White Paper, July 2001. 10, 32, 45, 80, 96

[94] Patrick D. O'Brien, Daniel C. Halbert, and Michael F. Kilian. The Trellis programming environment. In *Proceedings of the Conference on Object-oriented Programming, Systems, and Applications*, pages 91–102. ACM Press, New York, NY, USA, October 1987. 10, 95

[95] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, Urbana-Champaign, IL, USA, 1992. 87

[96] Harold Ossher and Peri Tarr. Hyper/J: Multi-dimensional separation of concerns for Java. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 734–737. ACM Press, New York, NY, USA, May 2000. 4

[97] Harold Ossher and Peri Tarr. *Multi-Dimensional Separation of Concerns and the Hyperspace approach*, volume 648 of *The Kluwer International Series in Engineering and Computer Science*, chapter 10. Kluwer Academic Publishers, Dordrecht, The Netherlands, 2000. 4

[98] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–194. ACM Press, New York, NY, USA, April 1984. 96

[99] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972. 3, 4

[100] David L. Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, 5(2):128–138, March 1979. 5

[101] David L. Parnas. Sofware aging. In *Proceedings of the 16th International Conference on Software Engineering*, pages 279–287. IEEE Computer Society Press, Los Alamitos, CA, USA, May 1994. 2, 5, 92

[102] Doron A. Peled. *Software Reliability Methods*. Springer-Verlag, Heidelberg, Germany, 2001. 17

[103] Shari Lawrence Pfleeger. Design and analysis in software engineering—part 1: The language of case studies and formal experiments. *Software Engineering Notes*, 19(4):16–20, October 1994. 47

[104] Alejandro Ramirez, Philippe Vanpeperstraete, Andreas Rueckert, Kunle Odutola, and Jeremy Bennett. *ArgoUML User Manual: A tutorial and reference description of ArgoUML, version 0.10*, May 2002. 67

[105] Awais Rashid, Ana Moreira, and João Araújo. Modularisation and composition of aspectual requirements. In *Proceedings of the 2nd International Conference on Aspect-oriented Software Development*, pages 11–20. ACM Press, New York, NY, USA, March 2003. 99

[106] Steven P. Reiss. Simplifying data integration: The design of the Desert software development environment. In *Proceedings of the 18th International Conference on Software Engineering*, pages 398–407. IEEE Computer Society Press, Los Alamitos, CA, USA, March 1996. 101

[107] Jason E. Robbins, David M. Hilbert, and David F. Redmiles. Extending design environments to software architecture design. In *Proceedings of the 11th Knowledge-Based Software Engineering Conference*, pages 63–72. IEEE Computer Society Press, Los Alamitos, CA, USA, September 1996. 67

[108] Jason E. Robbins, David M. Hilbert, and David F. Redmiles. Argo: a design environment for evolving software architectures. In *Proceedings of the 19th International Conference on Software Engineering*, pages 600–601. ACM Press, New York, NY, USA, May 1997. 67

[109] Martin P. Robillard. FEAT: An Eclipse plug-in for locating, describing, and analyzing concerns in source code. http://www.cs.ubc.ca/labs/spl/projects/feat, 2003. 33

[110] Martin P. Robillard. The Jex home page. http://www.cs.ubc.ca/spider/mrobilla/jex, 2003. 53

[111] Martin P. Robillard and Gail C. Murphy. Analyzing exception flow in Java programs. In *Proceedings of the 7th European Software Engineering Conference and 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 1687 of *Lecture Notes in Computer Science*, pages 322–337. Springer-Verlag, Heidelberg, Germany, September 1999. 18, 31, 53

[112] Martin P. Robillard and Gail C. Murphy. Concern Graphs: Finding and describing concerns using structural program dependencies. In *Proceedings of the 24th International Conference on Software Engineering*. ACM Press, New York, NY, USA, May 2002. 46, 48, 88, 89

[113] Martin P. Robillard and Gail C. Murphy. Automatically inferring concern code from program investigation activities. In *Proceedings of the 18th International Conference on Automated Software Engineering*. IEEE Computer Society Press, Los Alamitos, CA, USA, October 2003. To appear. 75

[114] Martin P. Robillard and Gail C. Murphy. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Transactions on Software Engineering and Methodology*, 2003. To appear. 18, 31

[115] Martin P. Robillard and Gail C. Murphy. A study of program evolution involving scattered concerns. Technical Report TR-2003-06, Department of Computer Science, University of British Columbia, Vancouver, BC, Canada, March 2003. 61, 75, 82, 86, 89

[116] Marc J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, 1(4):365–370, December 1975. 10

[117] H. Dieter Rombach and Bradford T. Ulery. Improving software maintenance through measurement. *Proceedings of the IEEE*, 4(77):581–595, April 1980. 1

[118] Barbara G. Ryder. Dimensions of precision in reference analysis of object-oriented programming language. In *Proceedings of the 12th International Conference on Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 126–137. Springer-Verlag, Heidelberg, Germany, April 2003. 30

[119] M. Sanella. *The Interlisp-D Reference Manual*. Xerox Corporation, Palo Alto, CA, USA, 1983. 10

[120] Carl F. Schaefer and Gary N. Bundy. Static analysis of exception handling in Ada. *Software—Practice and Experience*, 23(10):1157–1174, October 1993. 18

[121] Gunther Schmidt and Thomas Ströhlein. *Relations and Graphs—Discrete Mathematics for Computer Scientists*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Heidelberg, Germany, 1993. 119

[122] Robert Sedgewick. *Algorithms in C*. Addison-Wesley Publishing Company, Reading, MA, USA, 1990. 3

[123] Michael Siff and Thomas Reps. Identifying modules via concept analysis. *IEEE Transactions on Software Engineering*, 25(6), November/December 1999. 99

[124] Janice Singer. Practices of software maintenance. In *Proceedings of the International Conference on Software Maintenance*, pages 139–145. IEEE Computer Society Press, Los Alamitos, CA, USA, November 1998. 1

[125] Saurabh Sinha and Mary Jean Harrold. Analysis of programs with exception-handling constructs. In *Proceedings of the International Conference on Software Maintenance*, pages 348–357. IEEE Computer Society Press, Los Alamitos, CA, November 1998. 18, 31

[126] Gregor Snelting. Concept analysis—a new framework for program understanding. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 1–10. ACM Press, New York, NY, USA, June 1998. 98

[127] Elliot Soloway, Jeannine Pinto, Stan Letovsky, David Littman, and Robin Lampert. Designing documentation to compensate for delocalized plans. *Communications of the ACM*, 31(11):1259–1267, November 1988. 2, 100

[128] Ian Sommerville. *Software Engineering*. Addison-Wesley, 6th edition, 2001. 1

[129] Amie L. Souter, David Shepherd, and Lori L. Pollock. Testing with respect to concerns. In *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society Press, Los Alamitos, CA, USA, September 2003. To appear. 92

[130] Richard M. Stallman. EMACS the extensible, customizable self-documenting display editor. In *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation*, pages 147–156. ACM Press, New York, NY, USA, June 1981. 96

[131] Margaret-Anne D. Storey, Kenny Wong, and Hausi A. Müller. Rigi: A visualization environment for reverse engineering. In *Proceedings of the 19th International Conference on Software Engineering*, pages 606–607. ACM Press, New York, NY, USA, May 1997. 96

[132] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley Longman, Inc., Reading, MA, 2nd edition, 1991. 96

[133] Peri Tarr and Lori A. Clarke. Consistency management for complex applications. In *Proceedings of the 20th International Conference on Software Engineering*, pages 230–239. IEEE Computer Society Press, Los Alamitos, CA, USA, May 1998. 102

[134] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton. N degress of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering*, pages 107–119. IEEE Computer Society Press, Los Alamitos, CA, USA, May 1999. 3, 4

[135] Richard N. Taylor, Frank C. Belz, Lori A. Clarke, Leon Osterweil, Richard W. Selby, Jack C. Wileden, Alexander L. Wolf, and Michael Young. Foundations for the Arcadia environment architecture. In *Proceedings of the 3rd ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 1–13. ACM Press, New York, NY, USA, November 1988. 102

[136] Warren Teitelman and Larry Masinter. The Interlisp programming environment. *IEEE Computer*, 14(4):25–33, April 1981. 95

[137] Walter F. Tichy. RCS: A system for version control. *Software—Practice and Experience*, 15(7):637–654, July 1985. 10

[138] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995. 10, 96

[139] Frank Tip, Jong-Deok Choi, John Field, and G. Ramalingam. Slicing class hierarchies in C++. In *Proceedings of the Conference on Object-oriented Programming, Systems, and Applications*, pages 179–197. ACM Press, New York, NY, USA, October 1996. 96

[140] P. Tonella, G. Antoniol, R. Fiutem, and E. Merlo. Variable precision reaching definitions analysis for software maintenance. In *Proceedings of the 1st Euromicro Conference on Software Maintenance and Reengineering*, pages 60–67. IEEE Computer Society Press, Los Alamitos, CA, USA, March 1997. 31, 97

[141] Paolo Tonella. Concept analysis for module restructuring. *IEEE Transactions on software engineering*, 27(4), April 2001. 99

[142] Arie van Deursen and Tobias Kuipers. Identifying objects using cluster and concept analysis. In *Proceedings of the 21st International Conference on Software Engineering*, pages 246–255. IEEE Computer Society Press, Los Alamitos, CA, USA, May 1999. 99

[143] Robert J. Walker, Gail C. Murphy, Bjorn Freeman-Benson, Darin Wright, Darin Swanson, and Jeremy Isaak. Visualizing dynamic software system information through high-level models. In *Proceedings of the Conference on Object-oriented Programming Systems, Languages, and Applications*, pages 271–283. ACM Press, New York, NY, USA, October 1998. 48

[144] Robert J. Walker, Gail C. Murphy, Jeffrey Steinbok, and Martin P. Robillard. Efficient mapping of software system traces to architectural views. In *Proceedings of the 10th Annual IBM Centers for Advanced Studies Conference*, pages 31–40. IBM Corporation, Toronto, ON, Canada, 2000. 49

[145] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, July 1982. 96

[146] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984. 10, 96, 97

[147] Norman Wilde, Michelle Buckellew, Henry Page, Vaclav Rajlich, and LaTreva Pounds. A comparison of methods for locating features in legacy software. *Journal of Systems and Software*, 65(2):105–114, February 2003. 98

[148] Norman Wilde and Christopher Casey. Early field experience with the Software Reconnaissance technique for program comprehension. In *Proceedings of the International Conference on Software Maintenance*, pages 312–318. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996. 98

[149] Norman Wilde, Juan A. Gomez, Thomas Gust, and Douglas Strasburg. Locating user functionality in old code. In *Proceedings of the Conference on Software Maintenance*, pages 200–205. IEEE Computer Society Press, Los Alamitos, CA, USA, November 1992. 11, 97

[150] Norman Wilde and Michael C. Scully. Software reconnaissance: Mapping program features to code. *Software Maintenance: Research and Practice*, 7:49–62, 1995. 11, 97, 98

[151] W. Eric Wong, Swapna S. Gokhale, Joseph R. Horgan, and Kishor S. Trivedi. Locating program features using execution slices. In *Proceedings of the 1999 IEEE Symposium on Application-Specific Systems and Software Engineering and Technology*, pages 194–203. IEEE Computer Society Press, Los Alamitos, CA, USA, March 1999. 98

[152] Steven Woods and Qiang Yang. The program understanding problem: Analysis and a heuristic approach. In *18th International Conference on Software Engineering*, pages 6–15. IEEE Computer Society Press, Los Alamitos, CA, USA, March 1996. 1

[153] S.S. Yau and J.S. Collofello. Some stability measures for software maintenance. *IEEE Transactions on Software Engineering*, 6(6):545–552, November 1980. 1

[154] Robert K. Yin. *Case Study Research: Design and Methods*, volume 5 of *Applied Social Research Methods Series*. Sage Publications Ltd., London, UK, 2nd edition, 1989. 47, 48, 52

Appendix A

Relational Algebra

This appendix presents the notation and definitions of relational algebra used in the presentation of the formalisms. The notation and the definition of most relational operators are taken from Schmidt and Ströhlein [121]. Additional operator definitions are obtained from the presentation of Ince [61]. In this section it is assumed the reader is familiar with the basic concepts of set theory.

A.1 Notational Conventions

In this dissertation, the following notational conventions are used:

- Variables and label names are set in *italics* (e.g., the set S, the element e, the relation *Calls*).
- The names of entities found in source code and in windows of graphical user interfaces are set in `courier` type (e.g., class `A`, method `log()`, menu `File | Save As`).
- The names of mathematical functions are set in normal type (e.g., the range function, ran()).

The following additional conventions are used when referring to entities in Java programs.

- The name of classes are in lower-case letters, with the first letter of each word capitalized (e.g., `ChangeListener`).
- Then name of class members (fields and methods) begin with a lower-case letter, with the first letter of each following word capitalized (e.g., `firstItem`).

A.2 Definitions

Definition A.1 (Homogeneous Relation) *Let V be a set. A homogeneous relation R on V is a subset of the Cartesian product $V \times V$. Elements $x, y \in V$ are said to be in relation R if $(x, y) \in R$.*

Relations will usually be named, and defined either exhaustively by listing the corresponding set of pairs, or through a comprehensive specification (e.g., $\mathit{GreaterThan} = \{(x, y) | x > y\}$. When the underlying set V a relation is defined over is not specified by the context, it will be indicated as a subscript of the relation name (e.g., R_V).

Three special relations need to be considered: the empty relation, the identity relation, and the universal relation.

The empty relation denotes the absence of a relation between any elements of a set. It is represented with the symbol $\mathcal{O}$.

Definition A.2 (Empty Relation) *Let V be a set. $\mathcal{O}_V = \{\}$.*

The identity relation $\mathcal{I}$ puts every element in relation with itself.

Definition A.3 (Identity Relation) *Let V be a set. $\mathcal{I}_V = \{(x, y) \subseteq V \times V \mid x = y\}$.*

Finally, the universal relation $\mathcal{U}$ puts every element of a set in relation with every other element.

Definition A.4 (Universal Relation) *Let V be a set. $\mathcal{U}_V = V \times V$.*

We following definitions provide useful operators on relations.

Definition A.5 (Transpose) *Let $R \subseteq V \times V$ be a relation. We define the transpose of R, $R^\top$, as*

$$R^\top = \{(x, y) \in V \times V \mid (y, x) \in R\}.$$

Definition A.6 (Composition) *Let $R, S \subseteq V \times V$ be relations. Their composition $R \circ S \subseteq V \times V$ is given by*

$$R \circ S = \{(x, z) \in V \times V \mid \exists\, y \in V : (x, y) \in R \;\wedge\; (y, z) \in S\}.$$

Definition A.7 (Domain) *Let $R \subseteq V \times V$. The domain of R is the set*

$$\mathrm{dom}(R) = \{x \in V \mid (x, y) \in R\}$$

Definition A.8 (Range) *Let $R \subseteq V \times V$. The range of R is the set*

$$\mathrm{ran}(R) = \{y \in V \mid (x, y) \in R\}$$

Definition A.9 (nth Iterate) *The nth iterate of a relation, R^n, is its nth composition with itself.*

$$R^0 = \mathcal{I}, R^n = R \circ R^{n-1}$$

Definition A.10 (Reflexive Transitive Closure) *the reflexive transitive closure R^* of a relation R is the union of all its iterates*

$$R^* = R^0 \cup R^1 \cup R^2 \cup ...R^n.$$

Definition A.11 (Non-reflexive Transitive Closure) *The non-reflexive transitive closure R^+ of a relation R is the union of all its iterates except R^0*

$$R^+ = R^1 \cup R^2 \cup R^3 \cup ...R^n.$$

Definition A.12 (Domain Restriction) *The domain restriction operator $\lhd$ restricts the domain of a relation. It has two operands: the first operand is a set S; the second operand is a relation R. The result of the domain restriction operator is the subset of R which only contains pairs whose first element is contained in S:*

$$S \lhd R = \{(x, y) \in R \mid x \in S\}$$

We can also restrict the range of a relation.

Definition A.13 (Range Restriction) *Let S be a set and R be a relation.*

$$R \rhd S = \{(x, y) \in R \mid y \in S\}$$

Appendix B

Relations in Java Programs

This appendix defines the boolean functions on elements of a Java program used in Figure 3.1. The definitions are based on the Java Language Specifications [48]. In the rest of this appendix, references to the Java language specifications we be denoted by a simple subsection reference.

IsAClass(x) The function returns true if x represents a class declaration (§8.1).

IsAnInterface(x) The function returns true if x represents an interface declaration (§9.1).

IsAField(x) The function returns true if x represents a field declaration. Fields can be declared within a class declaration (§8.3), or as constants in an interface declaration (§9.3).

IsAMethod(x) The function returns true if x represents any declarative entity containing executable code or that can be dispatched to executable code. This definition thus encompasses methods declarations (§8.4), including abstract (§9.4,§8.4.3.1) and static (§8.4.3.2) method declarations, constructor declarations (§8.8), and instance (§8.6) and static (§8.7) initializer blocks. Although not explicitly represented in Java programs, default constructors (§8.8.7) can also be represented in a program model, and are considered to be methods. Instance field initialization code (§8.3.2.1) is considered to be included in each constructor for the class declaring the field (including the default constructor if applicable). Class field initialization code (§8.3.2.2) is considered to be part of the static initializer block for the class. A default static initializer block can be defined for this purpose if necessary.

Accesses(x,y) The function returns true if x is a concrete (non-abstract) method, y is a field, and x contains a field access expression (§15.11) referring to field y.

Calls(x,y) The function returns true if x is a concrete (non-abstract) method, y a concrete or abstract method (defined in a class or interface), and x contains a method invocation expression (§15.12) such that:

1. y is the compile-time method determined by the algorithms of (§15.12.1-§15.12.3), or
2. y is a valid runtime binding for the compile-time method determined above.

In other words, Calls(x,y) puts in relation a method and both the static method binding and all the potential dynamic bindings for the static binding. Determining which potential dynamic bindings are applicable is dependent on a specific implementation the static analysis algorithm for extracting the model, such as Class Hierarchy Analysis [32], or Rapid Type Analysis [5].

Checks(x,y) The function returns true if x is a concrete (non-abstract) method, y is a non-primitive type (class or interface), and x contains code containing a cast expression (§15.16) naming type y, or an `instanceof` operation (§15.20.2) naming type y.

Creates(x,y) The function returns true if x is a concrete (non-abstract) method, y is a class type, and x contains a class instance creation expression (§15.9) naming type y.

Declares(x,y) The function returns true if y is declared directly within the declaration of x. The following relations are possible:

- A type (class or interface) can declare fields (§8.3,§9.3) or methods (§8.4,§9.4);
- A type (class or interface) can declare another type (member class §8.5, or member interface §9.5).
- A concrete (non-abstract) method can declare a class (local class, §14.3, or anonymous class, §15.9.5).

ExtendsClass(x,y) The function returns true if x and y are both class types, and x is declared to extend y (i.e., directly extends y, §8.1.3).

ExtendsInterface(x,y) The function returns true if x and y are both interface types, and x is declared extend y (i.e., directly extends y, §9.1.2).

HasParameterType(x,y) The function returns true if x is a method (abstract or concrete), y is a non-primitive type, and y is contained in the list of parameters of x (§8.4.1).

HasReturnType(x,y) The function returns true if x is a non-constructor, non-initializer method (abstract or concrete), y is a non-primitive, non-void type, and y is the return type of x (§8.4).

Implements(x,y) The function returns true if x is a class type, y an interface type, and x declares to implement y (i.e., directly implements y, §8.1.4).

OfType(x,y) The function returns true if x is a field, y a non-primitive type (class or interface), and x is declared to be of type y (§8.3,§9.3).

Overrides(x,y) The function returns true if x is a concrete method, y is a method (concrete or abstract), and x overrides y (§8.4.6.1).

Appendix C

Transcripts for the jEdit Case Study

This appendix contains the partial transcripts of the jEdit case study relevant to the investigation and implementation of requirement 5 of the modification request (see Section 4.5.2). The transcripts list the actions performed by each subject for all of the episodes involving the discovery or use of information related to the methods `Buffer.load(View,boolean)` and `Buffer.recoverAutosave(View)`.

A transcript consists of a list of user *actions*. An action is a record consisting of four fields. In the transcript, an action appears as a line, with each field presented in a separate column. The first field contains the time of the action, in terms of elapsed seconds since the beginning of the study phase (investigation or execution). The second field contain the Eclipse view in which the event was triggered. Table C.1 lists the possible values for this field.

The third field in a transcript action describes the action performed by the subject. Table C.2 lists the possible values for this field. Finally, the fourth field for an action lists the target of the action.

The sections of the transcripts presented in this appendix list all actions for which the target is either method `Buffer.load(...)` or method `Buffer.recoverAutosave(...)`.[1], and the five actions preceding or following a relevant action. In the transcripts, actions for which the target includes either of the relevant methods are set in boldface.

[1]For a more concrete presentation, we have replaced the list of parameter types in method signatures by the symbol "...".

Table C.1: View codes

Code	**Description**
Concerns	The FEAT Concern Graph View.
Editor	A text editor or the editor area of a perspective.
Explorer	The Eclipse Package Explorer View.
Participants	The FEAT Participants View.
Projection	The FEAT Projection View.
Relations	The FEAT Relations View.
Result	A view listing Eclipse search results.
Tasks	An Eclipse View listing a set of tasks (e.g., syntax errors).
Viewer	An HTML Browser (e.g., Internet Explorer).
Workbench	The Eclipse workbench toolbar or menus.

Table C.2: Action codes

Code	Description
Add	Add an element to a concern.
Browse	Peruse the declarative structure of a class or source code.
Change	Modify an element.
Execute	Execute (i.e., run) an application).
New	Create a new element.
Query	Perform a structural (i.e., cross-reference) query.
Recall	Make visible an element previously accessed.
Search	Perform a lexical (i.e., keyword) search.
Select	Select an element in a view, when this actions has a side effect.
View	Access the source code for an element.

C.1 Subject C1

Investigation

```
0:38:55 Explorer Browse Buffer
0:38:59 Explorer View   Buffer.save(...)
0:39:01 Explorer View   Buffer.save(...)
0:39:04 Editor   Browse Buffer.java
0:39:54 Explorer Browse Buffer
0:39:59 Explorer View   Buffer.recoverAutosave(...)
0:40:35 Explorer Browse Buffer
0:40:45 Explorer View   Buffer.autosave()
0:41:14 Explorer Browse Buffer
0:41:18 Explorer View   Buffer.saveAs(...)
0:41:20 Explorer View   Buffer.save(...)
 ...
0:45:00 Result   View   Buffer.autosaveFile referenced by  Buffer.close()
0:45:10 Editor   Change Notes (not relevant)
0:45:35 Result   View   Buffer.autosaveFile referenced by Buffer.finishSaving(...)
0:45:38 Result   View   Buffer.autosaveFile referenced by Buffer.getAutosaveFile()
0:45:43 Result   View   Buffer.autosaveFile referenced by Buffer.setPath(...)
0:45:59 Result   View   Buffer.autosaveFile referenced by Buffer.recoverAutosave(...)
0:46:05 Result   View   Buffer.autosaveFile referenced by Buffer.load(...)
0:46:06 Result   View   Buffer.autosaveFile referenced by Buffer.finishSaving(...)
0:46:14 Result   View   Buffer.autosaveFile referenced by Buffer.close()
0:46:20 Result   View   Buffer.autosaveFile referenced by Buffer.autosave()
0:46:23 Result   View   Buffer.autosaveFile referenced by Buffer.finishSaving(...)
0:46:27 Editor   Browse Buffer.java
 ...
0:48:41 Explorer View   jEdit.propertiesChanged()
0:49:08 Editor   Change Notes (not relevant)
0:50:14 Editor   Change Notes (not relevant)
0:53:08 Editor   Change Notes (relevant)
0:53:21 Explorer Browse jEdit
0:53:33 Result   View   Buffer.autosaveFile referenced by Buffer.recoverAutosave(...)
0:53:44 Editor   Change Notes (recoverAutosave, wrong name)
0:53:58 Editor   Change Notes (recoverAutosave, corrects the name)
0:54:07 Editor   Browse jEdit
0:54:20 Editor   Change Notes (implementation strategy for R5)
0:55:53 Editor   Query  Buffer.recoverAutosave(...) referenced by
0:55:56 Result   View   Buffer.recoverAutosave(...) referenced by Buffer.load(...)
0:57:14 Editor   Recall Autosave.java
0:57:53 Editor   View   Notes
0:58:15 Editor   Change Notes (not relevant)
```

Execution

```
 ...
0:56:13 Workbench Execute jEdit
0:56:30 Editor    View    Notes (R5).
0:56:48 Editor    Recall  Buffer.java
0:56:54 Explorer  Browse  Buffer
0:56:59 Explorer  View    Buffer.removeAllMarkers()
0:57:00 Explorer  View    Buffer.recoverAutosave(...)
0:57:05 Explorer  Change  Buffer.recoverAutosave(...)
0:57:34 Workbench Execute jEdit
1:00:33 Editor    Recall  jEdit.java
1:00:42 Editor    Change  jEdit.propertiesChanged()
```

C.2 Subject C2

Investigation

```
 ...
0:16:16 Editor   Change Notes (not relevant)
0:16:30 Editor   Recall Autosave.java
0:17:33 Editor   Browse Autosave.java
0:17:57 Editor   View   Buffer.autosave()
0:18:40 Explorer Browse Buffer
0:19:08 Explorer View   Buffer.recoverAutosave(...)
0:19:29 Explorer Browse Buffer
0:19:32 Editor   View   Notes
0:19:42 Editor   Recall Autosave.java
0:19:44 Editor   Recall Buffer.java
0:20:13 Editor   Change Notes (requirement 3)
 ...
0:23:51 Editor   View   File.delete()
0:23:56 Editor   Recall Buffer.java
0:24:02 Explorer View   Buffer.autosave
0:24:21 Editor   Seach  ‘‘delete’’ in Buffer.java
0:24:40 Editor   Search "autosaveFile.delete" in Buffer.java
0:24:47 Editor   Browse Buffer.load(...)
0:25:12 Editor   View   File.delete()
0:25:18 Editor   Recall Buffer.java
0:25:28 Editor   Query  File.delete() referenced by
0:26:05 Editor   Search "autosaveFile.delete" in Buffer.java
0:26:25 Editor   Browse Buffer.java
0:26:51 Editor   Browse Buffer.load(...)
0:26:59 Editor   View   Notes
0:27:54 Editor   Recall Autosave.java
0:28:04 Editor   Recall LoadSaveOptionPane.java
0:28:08 Editor   Browse LoadSaveOptionPane.java
0:28:28 Editor   Recall Autosave.java
 ...
0:41:50 Editor   Recall Buffer.java
0:41:53 Explorer Browse jEdit
0:42:00 Explorer View   jEdit.openFile(...)
0:42:04 Editor   Recall Buffer.java
0:42:09 Explorer Browse Buffer.java
0:42:12 Explorer View   Buffer.recoverAutosave(...)
0:42:34 Editor   Recall jEdit.java
0:42:37 Editor   Browse jEdit.java
0:42:50 Explorer Browse jEdit
0:42:53 Explorer View   jEdit.propertiesChanged()
0:43:03 Explorer Browse jEdit
 ...
```

```
0:43:03 Explorer Browse jEdit
0:43:15 Explorer View   jEdit.propertiesChanged()
0:43:21 Explorer Recall Buffer.java
0:43:28 Editor   Browse Buffer.java
0:43:30 Explorer Browse Buffer
0:43:35 Explorer View   Buffer.recoverAutosave(...)
0:43:50 Editor   Query  Buffer.recoverAutosave(...) referenced by
0:43:53 Results  View   Buffer.recoverAutosave(...) referenced by Buffer.load(...)
0:44:44 Editor   View   Notes
0:44:51 Editor   Recall Buffer.java
0:44:55 Editor   Recall Notes
0:44:58 Editor   Change Notes: (adds ‘‘buffer.load()’’)
0:45:04 Editor   Recall Buffer.java
0:45:07 Editor   Browse Buffer.java
0:45:15 Explorer Browse Buffer
0:45:17 Editor   Recall Notes
0:45:20 Editor   Change Notes (adds Buffer.recoverAutosave())
0:45:58 Editor   Recall Buffer.java
0:46:01 Explorer Browse Buffer
0:46:18 Explorer View   Buffer.save(...)
0:46:20 Explorer View   Buffer.saveAs(...)
0:46:21 Explorer View   Buffer.save(...)
 ...
0:49:33 Result   View   Buffer.finishSaving(...) references "delete"
0:49:37 Editor   Browse Buffer.java
0:50:15 Explorer Browse Buffer
0:50:22 Explorer View   Buffer.autosaveFile
0:50:25 Result   View   Buffer.finishSaving(...) references "delete"
0:50:50 Result   View   Buffer.load(...) references "delete"
0:51:12 Editor   Recall Notes
0:51:18 Editor   Recall Buffer.java
0:51:22 Editor   Recall Notes
0:51:26 Editor   Change Notes (adds reload to R5)
0:51:28 Editor   Recall Buffer.java
0:51:37 Editor   Query  Buffer.load(...) referenced by
0:51:46 Result   View   Buffer.load(...) referenced by Buffer.checkModTime(...)
0:51:57 Result   View   Buffer.load(...) referenced by Buffer.reload(...)
0:52:05 Result   View   Buffer.load(...) referenced by jEdit.openFile(...)
0:52:09 Result   View   Buffer.load(...) referenced by jEdit.openTemporary(...)
0:52:13 Result   View   Buffer.load(...) referenced by jEdit.reloadAllBuffers(...)
0:52:18 Explorer Browse Buffer
0:52:32 Explorer View   Buffer.load(...)
0:52:54 Editor   Recall Notes
0:53:02 Editor   Change Notes (add info for R5)
0:53:10 Editor   Recall Buffer.java
0:53:21 Editor   Recall Notes
0:53:43 Editor   Recall Buffer.java
 ...
```

Execution

```
 ...
0:17:32 Editor   Recall jEdit.java
0:17:42 Editor   Change jEdit.propertiesChanged()
0:19:07 Editor   Recall Notes
0:19:22 Explorer Browse jEdit
0:19:30 Explorer Browse Buffer
0:19:43 Explorer View   Buffer.load(...)
0:19:52 Editor   View   Buffer.recoverAutosave(...)
0:19:58 Editor   Change Buffer.recoverAutosave(...)
0:20:01 Editor   Recall LoadSaveOptionPane.java
0:20:07 Editor   Recall jEdit.java
```

```
0:20:12 Editor   Recall Buffer.java
0:20:14 Editor   Change Buffer.recoverAutosave(...)
0:20:46 Editor   Query  Buffer.recoverAutosave(...) referenced by
0:20:56 Editor   Recall Notes
0:21:05 Editor   Change Notes (relevant)
0:22:04 Editor   Recall Buffer.java
0:22:12 Editor   Browse Buffer.java
0:22:16 Explorer Browse Buffer
0:22:32 Explorer View   Buffer.recoverAutosave(...)
0:22:37 Editor   Browse Buffer.recoverAutosave(...)
0:22:39 Editor   Change Buffer.java
0:23:29 Editor   Search ‘‘autosavefile’’ in Buffer.java
0:23:39 Editor   Browse Buffer.java
0:23:40 Editor   Search ‘‘autosavefile’’ in Buffer.java
0:23:49 Editor   Browse Buffer.java
```

C.3 Subject F1

Investigation

```
 ...
0:41:12 Projection   View   jEdit.openFile(...) accessing BufferUpdate.CREATED
0:41:17 Projection   View   jEdit.openFile(...) accessing jEdit.bufferListLock
0:41:18 Projection   View   jEdit.openFile(...) accessing jEdit.saveCaret
0:41:26 Projection   Query  jEdit.openFile(...) calling
0:41:31 Projection   View   jEdit.openFile(...) calling BufferUpdate.$<$init$>$(...)
0:41:38 Projection   View   jEdit.openFile(...) calling Buffer.load(...)
0:41:41 Projection   View   Buffer.load(...)
0:41:52 Editor       Browse Buffer.load(...)
0:42:18 Projection   Query  Buffer.load(...) calling
0:42:23 Projection   View   Buffer.load(...) calling Buffer.recoverAutosave(...)
0:42:40 Concerns     New    Buffer recovery
0:42:50 Concerns     Select Buffer recovery
0:42:52 Projection   Add    Buffer.load(...) calling Buffer.recoverAutosave(...)
0:42:57 Projection   Query  Buffer.recoverAutosave(...) called by
0:43:00 Projection   View   Buffer.recoverAutosave(...) called by Buffer.load(...)
0:43:10 Projection   Query  Buffer.recoverAutosave(...) calling
0:43:13 Projection   View   Buffer.recoverAutosave(...) calling Buffer$4.$<$init$>$()
0:43:26 Editor       Browse Buffer.java
0:44:02 Projection   Query  Buffer declaring
0:44:12 Projection   View   Buffer.addBufferChangeListener(...)
0:44:22 Editor       Browse Buffer.java
0:44:32 Editor       View   Buffer.addMarker(...)
 ...
0:49:43 Projection   View   Buffer.setDirty(...) called by Buffer$1.run()
0:49:49 Projection   View   Buffer.setDirty(...) called by BufferOptions.ok()
0:50:03 Projection   Recall all of Buffer
0:50:08 Projection   Browse all of Buffer
0:50:20 Concerns     Select Buffer recovery
0:50:21 Participants Select Buffer.recoverAutosave(...)
0:50:25 Participants Query  Buffer.recoverAutosave(...) called by
0:50:30 Projection   View   Buffer.recoverAutosave(...) called by Buffer.load(...)
0:50:41 Editor       Browse Buffer.load(...)
0:51:16 Participants Select Buffer.recoverAutosave(...)
0:51:19 Relations    View   Buffer.recoverAutosave(...) called by Buffer.load(...)
0:52:01 Participants View   Buffer.recoverAutosave(...)
0:52:14 Relations    View   Buffer.recoverAutosave(...) called by Buffer.load(...)
0:52:29 Editor       Browse Buffer.load(...)
0:52:34 Participants Query  Buffer.load(...) called by
0:52:40 Projection   View   Buffer.load(...) called by Buffer.checkModTime(...)
0:52:50 Editor       Browse Buffer.checkModTime(...)
0:52:55 Projection   View   Buffer.load(...) called by Buffer.finishSaving(...)
```

```
0:52:59 Projection   View   Buffer.load(...) called by Buffer.reload(...)
0:53:11 Projection   View   Buffer.load(...) called by jEdit.reloadAllBuffers(...)
0:53:15 Editor       Browse jEdit.reloadAllBuffers(...)
0:53:46 Projection   View   Buffer.load(...) called by jEdit.openTemporary(...)
0:53:53 Projection   View   Buffer.load(...) called by jEdit.openFile(...)
0:54:04 Participants View   Buffer.load(...)
0:54:53 Editor       Browse Buffer.load(...)
0:55:14 Participants Query  Buffer declaring
0:55:18 Projection   Query  Buffer.autosaveFile accessed by
0:55:24 Projection   View   Buffer.autosaveFile accessed by Buffer.close()
0:55:32 Projection   View   Buffer.autosaveFile accessed by Buffer.finishSaving(...)
0:55:37 Projection   View   Buffer.autosaveFile accessed by Buffer.getAutosaveFile()
 ...
```

Execution

```
        ...
0:21:57 Participants View    LoadSaveOptionPane._save()
0:22:03 Viewer       View    Instructions
0:22:11 Editor       Recall  jedit_gui.props
0:22:17 Editor       Change  jedit_gui.props
0:22:50 Concerns     Select  Buffer recovery
0:22:53 Participants Select  Buffer.recoverAutosave(...)
0:22:57 Participants View    Buffer.load(...)
0:23:25 Editor       Browse  Buffer.load(...)
0:23:58 Participants Query   Buffer declaring
0:24:03 Projection   Browse  Buffer declaring
0:24:18 Tasks        Select  Syntax error in Buffer.java
0:24:29 Editor       Search  ‘‘LoadAu’’
0:24:35 Editor       Search  ‘‘delete()’’
0:24:54 Editor       Browse  Buffer.java
0:25:05 Tasks        Select  Syntax error in Buffer.java
0:25:09 Editor       Change  Buffer.java
0:25:19 Participants Select  Buffer.recoverAutosave(...)
0:25:24 Participants View    Buffer.recoverAutosave(...)
0:25:30 Participants Select  Buffer.load(...)
0:25:32 Relations    View    Buffer.load(...) calling Buffer.recoverAutosave(...)
0:26:05 Participants Select  Buffer.recoverAutosave(...)
0:26:13 Participants View    Buffer.recoverAutosave(...)
0:26:17 Editor       Change  Buffer.recoverAutosave(...)
0:27:50 Concerns     Select  Buffer autosaving
0:27:53 Participants View    Autosave.actionPerformed(...)
0:28:00 Concerns     Select  Option settings
0:28:04 Concerns     Select  Buffer autosaving
0:28:10 Editor       Browse  Autosave.java
 ...
1:47:31 Editor       Browse  Autosave.java
1:47:41 Editor       Change  Autosave.java
1:48:51 Editor       Browse  Autosave.java
1:50:09 Workbench    Execute jEdit
1:51:08 Concerns     Select  Buffer recovery
1:51:09 Participants Select  Buffer.recoverAutosave(...)
1:51:23 Editor       Change  Buffer.recoverAutosave(...) (fix bug)
1:52:05 Workbench    Execute jEdit
1:52:51 Editor       Recall  jedit.props
1:52:53 Editor       Search  ‘‘autosave’’
1:52:54 Editor       Change  jedit.props
1:53:08 Workbench    Execute jEdit
 ...
```

C.4 Subject F2

Investigation

```
0:22:48 Projection   Add     Buffer.getAutosaveFile()
0:23:07 Projection   Browse  Buffer
0:23:19 Projection   Query   Buffer.autosaveFile accessed by
0:23:23 Projection   View    Buffer.autosaveFile accessed by Buffer.close()
0:23:32 Projection   View    Buffer.autosaveFile accessed by Buffer.finishSaving(...)
0:23:56 Projection   View    Buffer.autosaveFile accessed by Buffer.recoverAutosave(...)
0:24:18 Projection   Add     Buffer.autosaveFile accessed by Buffer.recoverAutosave(...)
0:24:38 Projection   Query   Buffer.recoverAutosave(...) called by
0:24:42 Projection   View    Buffer.recoverAutosave(...) called by Buffer.load(...)
0:24:54 Projection   Add     Buffer.recoverAutosave(...) called by Buffer.load(...)
0:24:59 Editor       Browse  Buffer.load(...)
0:25:38 Editor       Browse  Buffer
0:25:44 Projection   View    Buffer.load(...)
0:25:55 Participants Select  Buffer.recoverAutosave(...)
0:25:58 Participants View    Buffer.load(...)
0:26:01 Relations    View    Buffer.load(...) calling Buffer.recoverAutosave(...)
0:26:11 Concerns     Select  Clean up backup files
0:26:12 Concerns     Select  Recover from backup
0:26:12 Concerns     Select  Make backup files
0:26:15 Viewer       View    Instructions
0:26:36 Participants Select  Buffer.autosave()
 ...
0:49:00 Participants Select  Autosave.actionPerformed(...)
0:49:03 Participants View    Autosave.actionPerformed(...)
0:49:55 Editor       Browse  Autosave.java
0:50:31 Concerns     Select  Recover from backup
0:50:35 Participants Select  Buffer.autosaveFile
0:50:38 Participants Select  Buffer.load(...)
0:50:42 Participants Select  Buffer.recoverAutosave(...)
0:50:48 Participants View    Buffer.recoverAutosave(...)
0:51:13 Participants Query   Buffer.recoverAutosave(...) called by
0:51:17 Projection   View    Buffer.recoverAutosave(...) called by Buffer.load(...)
0:53:02 Editor       Browse  Buffer.java
0:53:32 Concerns     Select  Clean up backup files
0:53:36 Participants Select  jEdit.getBuffers()
0:53:50 Participants Select  Autosave.actionPerformed(...)
0:53:52 Participants View    Autosave.actionPerformed(...)
 ...
```

Execution

```
 ...
0:13:12 Participants Select  Autosave.actionPerformed(...)
0:13:13 Participants View    Autosave.actionPerformed(...)
0:13:20 Editor       Recall  jEdit.propertiesChanged()
0:13:23 Editor       Change  jEdit.propertiesChanged()
0:15:17 Concerns     Select  Recover from backup
0:15:19 Participants Select  Buffer.recoverAutosave(...)
0:15:23 Participants View    Buffer.recoverAutosave(...)
0:15:32 Participants View    Buffer.load(...)
0:15:36 Editor       Browse  Buffer.load(...)
0:16:50 Participants Select  Buffer.recoverAutosave(...)
0:16:51 Participants View    Buffer.recoverAutosave(...)
0:17:01 Participants View    Buffer.load(...)
0:17:06 Editor       Browse  Buffer.load(...)
0:18:30 Editor       Change  Buffer.load(...)
0:19:45 Editor       Search  modTime
0:20:09 Participants View    Buffer.load(...)
```

```
0:20:20 Editor       Browse  Buffer.load(...)
0:20:49 Editor       Change  Buffer.load(...)
0:21:40 Participants View    Buffer.load(...)
0:21:43 Editor       Change  Buffer.load(...)
0:22:04 Concerns     Select  Make backup files
0:22:14 Viewer       View    Instructions
0:22:57 Workbench    Execute jEdit
0:26:09 Concerns     Select  Make backup files
0:26:11 Participants Select  Buffer.autosave()
 ...
```