NaCIN – An Eclipse Plug-In for Program Navigation-based Concern Inference

Imran Majid and Martin P. Robillard
School of Computer Science
McGill University
Montreal, QC, Canada
{imajid, martin} @cs.mcgill.ca

ABSTRACT
In this paper we describe NaCIN, an Eclipse plug-in that records a developer's code navigation activity and produces sets of elements potentially implementing different concerns relevant to the current task. It performs an analysis of the navigation paths and structural dependencies of the recorded elements and clusters the results in groups potentially associated with high level concepts. NaCIN partially automates the process of relating source code with high-level abstractions and enables knowledge about the implementation of different concerns to be reused in future investigations. We present the architecture and a preliminary assessment of NaCIN.

Categories and Subject Descriptors
D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.6 [**Software Engineering**]: Programming Environments; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms
Documentation, Experimentation, Human Factors

Keywords
Program Navigation, Program Investigation, Concern Inference, Concern Modeling

1. INTRODUCTION
Software projects typically require several modifications, often to add, modify, or enhance various features. In many cases an extensive investigation of the program is required to identify the code that is relevant to the change, especially when it is scattered across several modules. Integrated development environments (IDEs) can assist the developers in navigation and exploration of elements related to a certain feature or concern. Unfortunately, once code relevant to a change is discovered and understood, the information is usually not recorded. This results in a similar effort being expended in the case where any subsequent modifications of the same feature or concern have to be carried out by a different developer (or by the same developer if the system is large).

OOPSLA 2005 Eclipse Technology Exchange (ETX), October 16–17, 2005, San Diego, CA, USA.

Different approaches, such as Concern Graph [6], can be used to associate source code with high-level concepts such as features or bug reports. For example, using either FEAT [7] or ConcernMapper[1], a developer can build concern representations while exploring the source code, and save this information for future use. However, these techniques place the onus on the developers for producing the concern-to-code mapping.

The Navigation-based Concern Inference Eclipse plug-in (NaCIN) automates the concern generation activity: it monitors a developer's code navigation activity and produces an approximate concern model spanning the elements that have been explored. It provides the mechanism to record an exploration of the code and synthesizes this information for reuse in its subsequent investigations.

2. GUIDING PRINCIPLES
NaCIN analyzes a variety of relationships between elements navigated and uses this information to generate a concern model containing several concerns. This strategy differs from the direct computation of investigation frequencies for different code elements. Instead, our strategy is based on the following hypotheses:

- **Navigation paths can indicate high-level associations between program elements [8].** During the investigation of the source code, the probability that a certain element is associated with a given task can be determined by program investigation patterns, structural dependencies between elements investigated, and the kinds of event revealing the elements (e.g., an element that becomes visible during scrolling is likely to be less relevant than the one that is selected as a result of a cross-reference search).
- **A developer can look at everything visible, not only at selected elements.** NaCIN is based on the hypothesis that developers may look at code visible in the editor even if they have not selected it explicitly. Program elements related to a certain concern are generally placed in close proximity. For example, method declarations related to a concern are often placed together in a class. NaCIN provides a mechanism for recording all such elements that were not selected explicitly but still became visible to the developer.
- **The instrumentation required to collect fine-grained navigation events is not disruptive.** The monitoring of a developer's actions does not disrupt his work and does not

[1] http://www.cs.mcgill.ca/~martin/cm/

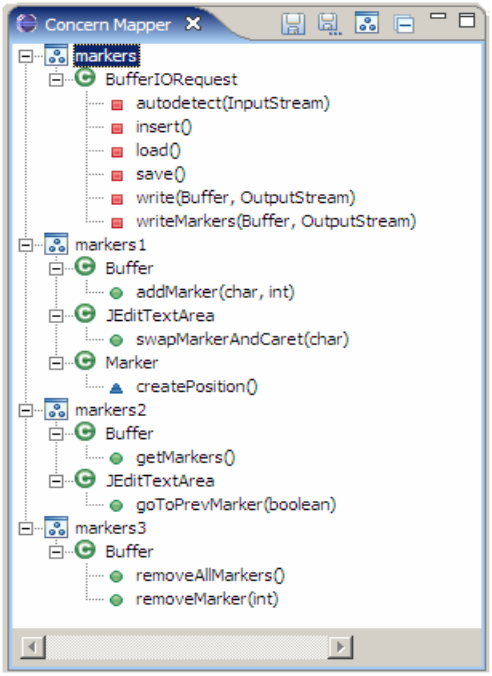
require any extra effort. It can be integrated seamlessly and does not slow down the machine to a point where it will undermine its utility.

- **The results are approximate.** Given the human aspect of the phenomenon analyzed, the results are expected to be approximate. As such, it should be possible and easy to modify the results to better reflect the implementation of concerns of interest.

3. SCENARIO

We describe NaCIN through a scenario of a software modification. In this scenario, a developer is asked to fix the problems related to the "Markers" concern of a popular open source text editor, jEdit[2] version 4.2. Markers are a means of highlighting certain lines in files and in version 4.2 some of the menu items related to this concern are not functioning as required. Before a developer can perform the required modifications, the developer has to identify the code related to this concern. The developer selects to record the transcript of his investigation activity by pressing the **Start** button of NaCIN. NaCIN now records information about the navigation of the developer through source code.

The developer starts by searching for the declaration of a "Marker" Java type in jEdit and finds one. The developer browses its code briefly and, to find the places where it is being used, searches for the references of its constructor in the project. The search leads to the `addMarker` method in the `Buffer` class. The developer scrolls around the `Buffer` class and finds a collection of methods related to markers. He uses the Search, Type Hierarchy and Call Hierarchy views repetitively to find other related methods scattered in several classes. During this investigation task, the developer also selects elements of significance from the Package Explorer and the Outline views.



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

Figure 1. Markers Concerns inferred by NaCIN

As this investigation focused on one feature of the system, Markers, the developer mostly visited the elements relevant to it. After spending some time investigating the source code, the developer believes he has some understanding of this concern and presses the **Generate** button of NaCIN. The developer provides the name of the concern and the number of elements it should have. NaCIN analyzes the investigation data and generates a concern model. This concern model is then made visible to the user through an Eclipse View (Figure 1). The generated concern model requires a final review by the developer, who adds those elements that have been missed by NaCIN and removes those that are not relevant to the Markers concern. This concern model can be saved as an XML file and can be used for future investigations related to the Markers concern of jEdit.

NaCIN, instead of putting all the navigated elements in one basket, clusters them into relevant concerns. In Figure 1, the *markers* concern represents the methods related to the storage mechanism of markers while *markers3* represents the methods related to the removal of markers. These concerns can be renamed and altered using the view shown in Figure 1.

4. ARCHITECTURE

NaCIN comprises three modules, The Recorder, The Inference Engine and ConcernMapper (the user interface). The Recorder records a developer's code navigation activity and generates an investigation transcript file in XML format. The Inference Engine performs inference on the investigation transcript and produces a Concern Model. This concern model is passed to ConcernMapper, which displays it to the user. ConcernMapper has been developed independently and allows a developer to store program elements relevant to a concern. In our architecture it is used as a third-party component.

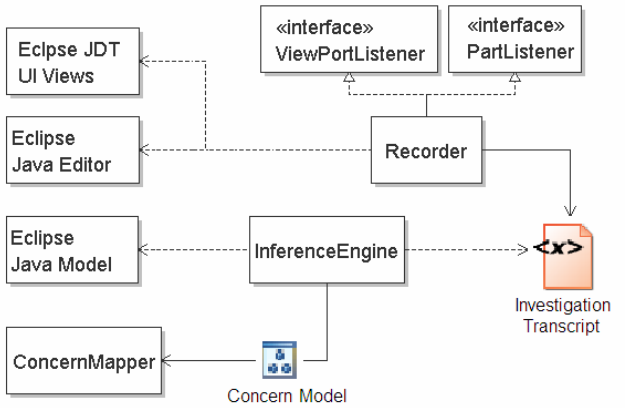


Figure 2. Architecture of NaCIN

4.1 The Recorder

The Recorder records the actions performed by the developer (e.g., selecting a method in the Package Explorer) and the corresponding visible information (i.e., all the methods with code visible in the editor). In terms of the visible information, method declaration is used as the unit of granularity. The Recoder runs seamlessly in the background and captures all the method declarations that become visible on the screen as a developer navigates the code using the tools provided by Eclipse, like

Package Explorer, Outline, Search, Type Hierarchy and Call Hierarchy views. The Recorder generates an XML transcript containing an ordered list of such events.

The recorder distinguishes between five different categories of events [5]:

- **Selection:** the content of the active editor changed as the result of selecting an element to be displayed
- **Cross-reference:** the content of the active editor changed as the result of following a cross-reference between two elements.
- **Recall:** an editor window was recalled from an existing buffer of visible windows, such as a history list or tabbed pane
- **Scrolling:** the content of the active editor window changed as the result of scrolling up and down in a file
- **Text Search:** the content of the active window changed as the result of selecting an element through a keyword search

The Recorder is implemented by instrumenting the `jdt.ui` plug-in of Eclipse. When an editor part is activated, a singleton class (`Recorder`) is added as its listener. The Recorder implements `PartListener`, which enables it to listen to activation/deactivation of editor parts and generates events accordingly. It also implements `ViewportListener` enabling it to record an event whenever the visible area of the text editors is changed.

The following locations are instrumented in `jdt.ui` (identified in Figure 3)

1. Events are recoded when elements are opened in an editor from the Package Explorer. Similar events are recorded when an element is opened using the short-cut key (F3).
2. Selection of an element in the Outline view generates an event.
3. Selection of an element from the Java Search Result Page generates a cross-reference event.
4. Selection of an element in the Type Hierarchy View generates an event.
5. Selection of an element in the Call Hierarchy View produces an event.
6. A text search event is generated on a keyword search in editor.

We are currently experimenting with different possible mappings between instrumentation points and event types.

4.2 The Inference Engine

The Inference Engine takes a transcript file as input and generates a group of elements that are potentially involved in the implementation of a concern. We use the algorithm described in a previous paper [5]. The algorithm can generate concerns based on a calculation of how different elements were related during program investigation session. The concern inference algorithm is divided into three phases. In the first phase, each visible element of every event is assigned a probability that this element was actually examined by the developer. In the second phase, a correlation metric of every pair of elements in the transcript is calculated. The Eclipse Java Model and Search Engine are used to identify the structural relationships between elements and the pairs of structurally connected elements are given proportionally more weight. The third phase generates a set of concerns based on the correlation metric calculated in the second phase and passes these to ConcernMapper.

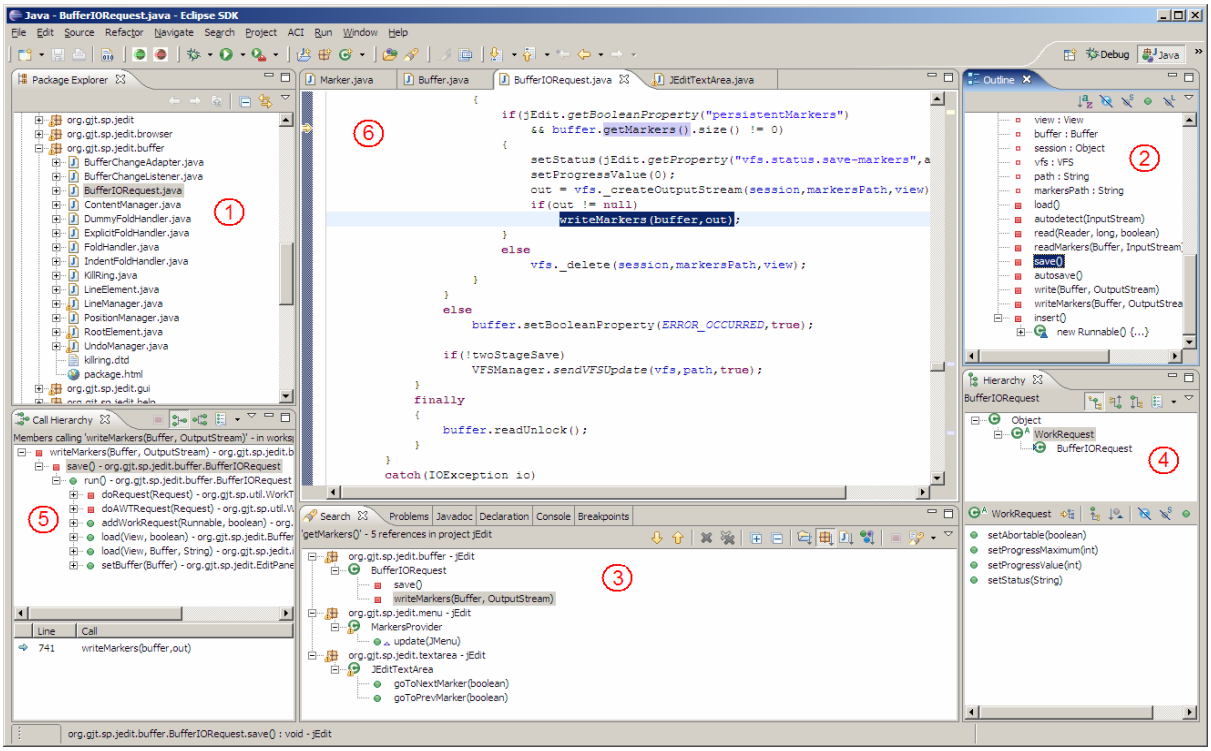


Figure 3. Instrumented views of Eclipse jdt

4.3 ConcernMapper

ConcernMapper is developed separately as an Eclipse plug-in. Developers can use it independently as they navigate the code and associate elements with a Concern Model. However in our case it takes in as input the elements of a concern generated by the Inference Engine. The integration with ConcernMapper is straightforward. We create a concern by giving its name and then add the elements in that concern. This integration allows the user to immediately analyze the concerns produced by the inference engine. The user can further prune and complete the representations identified by the algorithm using ConcernMapper. ConcernMapper also provides the facility to store the concern, allowing it to be used in later investigations.

5. PRELIMINARY EVALUATION

NaCIN is currently in an early phase of development and being assessed though small experiments. We present one of these experiments here as a way to illustrate the cost and benefits of using this technology. In this experiment, we asked three developers to investigate how they would improve a weakness in the implementation of Violet[3], a UML editor written in Java consisting of approximately 6600 lines of code. The change posited in this study concerned an inconsistency between associations supported by different UML figures. In this study, the subjects (S1, S2 and S3) were asked to investigate the code of Violet.

S1 is the second author of the paper. S2 is an undergraduate student with around 3 years of development experience. S3 is an undergraduate student with one year of development experience. None of the subjects are part of the development team of NaCIN (although S1 is obviously associated with the project). The subjects all had experience with Eclipse but had no previous knowledge of the code of Violet. They were not given any initial hint and were not allowed to use the debugger, though they could modify the program by inserting print statements.

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

- **Nodes and Edges:** A set of classes responsible for figures and edges displayed on different diagrams.
- **Connection:** The event-handling system that connects edges with nodes.

5.1 Results

All three subjects were able to locate and rectify the problem related to the given task. We applied our inference algorithm on each transcript, requesting in each case a concern model for 10 elements. Table 1 shows the characteristics of the transcripts produced. The second column lists the number of events, the third column lists the number of unique program elements visible to the developer during this investigation, the fourth column shows the number of classes navigated in this study and the fifth column shows the amount of time taken to complete the task.

[3] http://www.horstmann.com/violet/

Table 2. Characteristics of investigation transcripts

Subject	Events	Elements	Classes	Time
S1	40	62	8	12 mins
S2	91	74	11	20 mins
S3	79	104	20	30 mins

Tables 2, 3 and 4 show the concern models generated for subjects S1, S2 and S3 respectively.

Table 2. Results for Subject S1

	Concern Model

1	`Edge.getEnd()`
	`Edge.getStart()`
	`Graph.add(Node, Point2D)`
	`Graph.connect(Edge, Point2D, Point2D)`
	`NoteEdge.draw(Graphics2D)`
	`NoteEdge.getConnectionPoints()`
	`SegmentedLineEdge.getConnectionPoints()`
	`ShapeEdge.getShape()`

2	`EditorFrame.changeLookAndFeel(String)` `EditorFrame.EditorFrame(Class)`

Table 3. Results for Subject S2

	Concern Model

1	`Graph.findEdge(Point2D)`
	`Graph.findNode(Point2D)`
	`GraphPanel$1.mousePressed(MouseEvent)`

2	`Edge.connect(Node, Node)`
	`Graph.connect(Edge, Point2D, Point2D)`
	`NoteEdge.draw(Graphics2D)`
	`NoteEdge.getConnectionPoints()`
	`NoteEdge.getShape()`

3	`RectangularNode.writeObject(ObjectOutp..)` `RectangularNode.writeRectangularShape(Object..)`

Table 4. Results for Subject S3

	Concern Model
1	`CallNode.getConnectionPoint(Direction)` `NoteNode.NoteNode()`

2	`AbstractNode.AbstractNode()`
	`ReturnEdge.getPoints()`
	`NoteNode.getShape()`
	`ShapeEdge.getShape()`
	`NoteEdge.getConnectionPoints()`
	`RectangularNode.clone()`
	`ReturnEdge.ReturnEdge()`
	`NoteNode.draw(Graphics2D)`

Concern models generated for S1 and S2 included the most important elements and classes (`Graph, Edge, NoteEdge`) related to the assigned task. The methods in bold in Table 2 and 3 not only represent the common methods between the concern models of S1 and S2 but also the location where the modification was made. It is not surprising that NaCIN was able to identify the modified location along with the relevant elements since modified elements usually form an important part of the navigation activity. Clustering of relevant elements ensures that those navigated pieces of code that are heavily associated during the program navigation are reported as part of the same concerns. Concern 2 in case of S1 and Concern 3 in case of S2 represent this aspect of the approach.

Even though S3 was able to correct the problem, the randomness in his navigational activity made it impossible to identify any meaningful concern. This data reinforces a previous observation that the algorithm performs better in the case of a relatively methodical investigation of the code [5]. In these situations useful concerns can be documented at minimal cost to help in future tasks.

5.2 User Experiences and Feedback

The three subjects agreed that the presence of instrumented code was unnoticed and did not hinder their investigation of the code. The time required by the inference engine to generate the concern models was between 10 and 20 seconds. We are planning further empirical testing of the approach as an ongoing component of this research project.

6. RELATED WORK

FEAT allows the programmer to create views of structurally related elements by explicitly adding them to a Concern Graph [7]. The Concern Manipulation Environment also supports associating concerns with code through a query mechanism [2]. Both of these approaches use a specialized view to show the program elements related to the current task and place the burden on the programmer to declare the task-specific program elements. In contrast, NaCIN captures these program elements implicitly, reducing the programmer's effort.

Many IDE tools have monitored the programmer's context to present related program elements. Exton and McKeogh provide quantitative insights into how different programmers develop and maintain software [4]. They use Excel to visualize the recorded information. In comparison, NaCIN studies elements at finer granularity and analyzes the inter-relationships between them.

Mylar uses a degree-of-interest (DOI) model to capture the context of a task [3]. It focuses on the elements visible in IDE views but associates editing and navigation activity with program elements alone and does not model navigation paths. Instead, NaCIN is based on programming activity and infers the programmer's context by analyzing the structural navigation paths. NaCIN, by means of concern models, also provides storage and hence reusability of knowledge related to a task.

NavTracks keeps track of the navigation history of software developers, forming associations between related files [9]. These associations are then used as the basis for recommending potentially related files as a developer browses the software system. The value of the suggested files diminishes as the size of the files increases. Instead of focusing on file-to-file relationships, NaCIN works at the level of individual methods.

Several approaches have been developed to identify relevant source code using data mining. Zimmermann et al., have developed an approach that uses association rule mining on CVS data to recommend source code that is potentially relevant to a given fragment of source code [10]. Hipikat is a tool that provides recommendations about project information that a developer can consider during a modification task [1]. Hipikat draws its recommended information from a number of different sources apart from the source code. NaCIN can complement these approaches and the concern models generated by NaCIN will prove more helpful for data mining purposes.

7. CONCLUSIONS

NaCIN is an inexpensive tool that analyzes the developer's investigation of code and produces an approximate concern model spanning the explored elements. It records all the methods that become visible as the result of a navigation event and clusters them into groups based on navigation paths and structural relations. It partially automates the process of relating source code with high-level concerns and enables the information collected in this way to be reused in future investigations. An initial evaluation of NaCIN suggests that it can provide useful results in case of relatively methodical investigation of code.

8. ACKNOWLEDGMENTS

The authors are grateful to the study subjects. This research was supported by an Eclipse Innovation Award and by a start-up grant from McGill University.

9. REFERENCES

[1] Davor Cubranic and Gail C. Murphy. Hipikat: Recommending pertinent software development artifacts. *25th International Conference on Software Engineering (ICSE'03)*, pages 408–418, 2003.

[2] William Harrison, Harold Ossher, Stanley M. Sutton Jr., and Perri Tarr. *Concern Modeling in the Concern Manipulation Environment.* Research Report RC23344, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, September 2004.

[3] Mik Kersten and Gail C. Murphy. Mylar: a degree-of-interest model for IDEs. In *Proceedings of the 4th Conference on Aspect-Oriented Software Development*, pages 159–168, 2005.

[4] John McKeogh and Chris Exton. Eclipse plug-in to monitor the programmer behaviour. *OOPSLA, Eclipse Technology eXchange Workshop*, pages 93–97, 2004.

[5] 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*, pages 225–234, 2003.

[6] 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*, pages 406–416, 2002.

[7] Martin P. Robillard and Gail C. Murphy. FEAT: a tool for locating, describing, and analyzing concerns in source code. In *Proceedings of the 25th International Conference on Software Engineering*, pages 822–823. May 2003.

[8] Martin P. Robillard and Gail C. Murphy. Program navigation analysis to support task-aware software development environments. In *Proceedings of the ICSE Workshop on Directions in Software Engineering Environments*, pages 83–88. IEE, 2004.

[9] Janice Singer, Robert Elves, and Margaret-Anne Storey. NavTracks: Supporting navigation in software maintenance. In *Proceedings of the International Conference on Software Maintenance*, 2005. To appear.

[10] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering*, pages 563–572, 2004.