ConcernMapper: Simple View-Based Separation of Scattered Concerns

Martin P. Robillard and Frédéric Weigand-Warr

School of Computer Science
McGill University
Montreal, QC, Canada
{martin,fwwarr}@cs.mcgill.ca

ABSTRACT

We introduce ConcernMapper, an Eclipse plug-in for experimenting with techniques for advanced separation of concerns. ConcernMapper supports development and maintenance tasks involving scattered concerns by allowing developers to organize and view the code of a project in terms of high-level abstractions called *concerns*. ConcernMapper is also designed as an extensible platform intended to provide a simple way to store and query concern models created through a variety of approaches. This paper describes the user interface and internal architecture of ConcernMapper, and demonstrates how to write extensions for it.

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

Design, Documentation, Experimentation

Keywords

Separation of concerns, concern mapping, concern modeling, aspect-oriented software development

1. INTRODUCTION

An old tenet of software engineering tells us to design software systems to achieve "separation of concerns". This guideline is perfectly clear when heard a comfortable distance away from source code. Unfortunately, when we get down to details, troubling questions arise: What is a concern? (Is this design decision really a concern?) Who is this a concern for? (Current developers? Future maintainers? Users?) Will this always be a concern? (Are you really sure?)

Clearly, it is not possible to separate *all* concerns that may be of present or future interest to the various stakeholders of a software project. As a consequence, although the decomposition of a software system into modules can often be "good" in general, it is rarely "perfect" for a given software modification task. Typically, the code to understand and change in the context of a software modification task will cut across a number of modules (classes, files, etc.), and may participate in the implementation of various concerns (requirements, design decisions, etc.) Since previous research indicates that modifying the implementation of concerns whose code is not localized leads to particular challenges for software developers [6, 8], we are investigating techniques to allow developers to modify software systems in a way that naturally aligns with their concern of interest.

To provide a simple foundation for our research program on advanced separation of concerns, we developed an Eclipse plug-in to support a simple way to model concerns in source code. Our plug-in, ConcernMapper,[1] was developed with two goals in mind. First, we wanted ConcernMapper to support our daily software development activities by providing a simple way to model concerns. In an academic environment, where projects can experience a high personnel turnover and where a lot of the development is experimental, we wanted to provide Eclipse users with a very simple way to associate code with concerns in order to collect, share, and reuse this type of knowledge. As a second goal, we were interested in developing a simple platform for research on advanced separation of concerns. Many techniques can be used to automatically infer code that might be of interest to a developer, and there are countless ways to present this information back to developers. However, these ideas have one common denominator: they involve the association of code with high-level concerns. In designing ConcernMapper, we strove for an open architecture that can easily be extended by any researcher wishing to record information about the implementation of concerns using Eclipse. Section 4.2 gives an example of how ConcernMapper can be extended to automatically generate concerns based on a simple structural analysis of the program.

2. USING CONCERN MODELS

We describe the main features of our ConcernMapper plug-in through a scenario of software modification involving scattered concerns. Our scenario is taken from a previous study of the behavior of Eclipse developers grappling with a modification problem involving scattered concerns [8].

In this scenario, a developer using Eclipse with ConcernMapper is asked to enhance the "autosave" feature of a popular open-source text editor, jEdit.[2] The autosave feature in the jEdit text editor makes backups of all open files at a frequency that can be set

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

[2] www.jedit.org

by users. Not knowing how the autosave feature is implemented in jEdit or even where to start looking, the developer performs a general text search for the keyword “autosave”. This search yields 41 matches scattered over four Java files. The developer looks briefly at the files and decides that they indeed participate in the implementation of the autosave feature. Using ConcernMapper, the developer creates a concern called “Autosave feature”. The developer then tries to understand how the autosave feature is implemented by browsing and querying the code using the various JDT views (Package Explorer, Type Hierarchy, Call Hierarchy, etc.). Each time the developer finds elements relevant to the implementation of the autosave feature, they drag them into the Autosave concern in the ConcernMapper View. Figure 1 shows the ConcernMapper view at some point of the investigation.

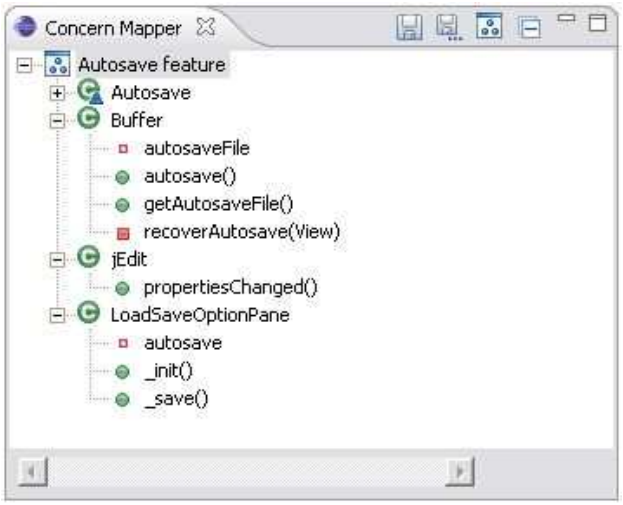

Figure 1: The ConcernMapper View with a single concern.

During the investigation, the developer is constantly informed of which elements are now part of his concern of interest. For example, in the results of any Java search, the subset of the results that are part of a concern are displayed in bold with the name of the concern (see Figure 2).

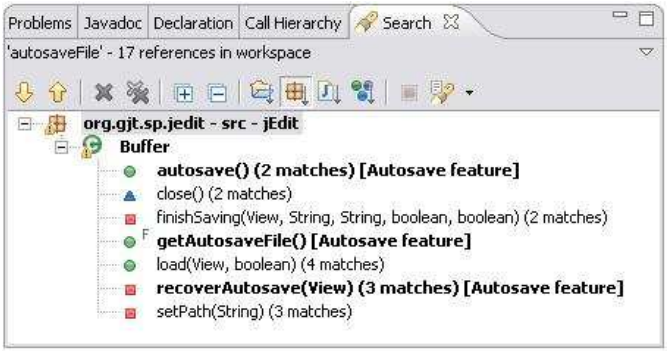

Figure 2: Search results with concern information.

After additional investigation, the developer understands the implementation of the autosave feature much better and discovers a natural separation between different aspects of the feature:

- The timing of the autosave event.
- The management of the state of a file (whether it was backed up since the last modification).
- The management of the option pane in the graphical user interface allowing users to change the frequency of autosave events.
- The code supporting the recovery from automatically saved backups.

By creating new concerns, renaming the existing one, and moving and copying elements between concerns in ConcernMapper, the developer quickly reorganizes the concern model to reflect a decomposition that will help solve the modification task (see Figure 3).

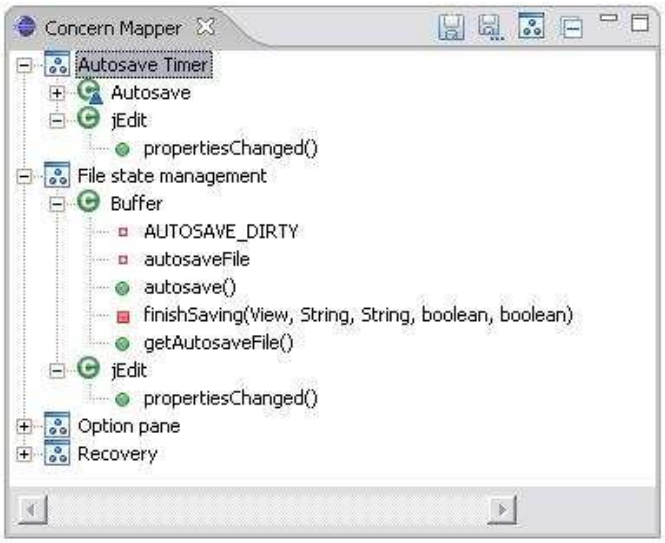

Figure 3: The ConcernMapper View with multiple concerns.

After spending some time investigating source code and understanding how the autosave feature is implemented, the developer realizes that it is already 8pm and probably a good time to call it a day. At this point, the developer saves the concern model (as an XML file) and adds it to the revision control system. The next morning (or week), when the developer has time to implement the change, the concern model is loaded into the ConcernMapper view, allowing the developer to immediately access the code relevant to each concern.

3. DESIGN AND IMPLEMENTATION

Although ConcernMapper can be useful as a stand-alone tool, the main motivation underlying its development was to provide a basic but extensible platform for experimenting with advanced separation of concerns mechanisms. To this end, we designed ConcernMapper for simplicity and ease of extension. This section describes the main architecture of ConcernMapper and the decisions motivating it.

Besides a basic plug-in class controlling its state, ConcernMapper is logically decomposed into two components: a model and a view. Figure 4 is a UML class diagram representing the key features of the architecture of ConcernMapper. Class `ConcernMapper` is the main plug-in class. It manages an instance of the `ConcernModel`. The `ConcernMapperView` is the class implementing an Eclipse view extension point. It can obtain a reference to the model through `ConcernMapper`, and can register itself as a listener for any changes to the model.

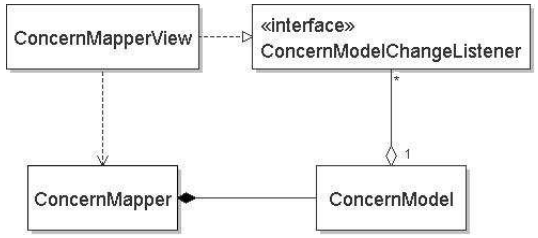

Figure 4: Key components of ConcernMapper.

3.1 The Concern Model Structure

The basic idea of ConcernMapper is to allow developers to associate parts of a program with high-level concerns. Ideally it should be possible to support the association of *any* part of a program with a concern, if this association can be useful to developers. Examples of program parts can include source code elements (e.g., methods, fields, classes, local variables, statements, comments) as well as fragments from other software engineering artifacts (e.g., UML model elements, individual requirements, sections from user manuals). To accommodate experimentation with such possibilities we have left our model open-ended and capable of supporting any type of element. The decision as to which element types to support is taken by the implementers of the model viewing components (in our case class `ConcernMapperView`). Section 3.3 describes and justifies the elements currently supported by ConcernMapper.

We express the structure of our concern model as a grammar using the extended Backus Naur form:

```
<model> ::= <concern>*
<concern> ::= <name><weighted-element>*
<weighted-element> ::= <object><degree>
```

As this specification states, a concern model consists of zero or more concerns, where each concern maps a name to zero or more weighted elements. A weighted element is simply a pair associating an object of unspecified type with a value indicating the membership degree of the object in the concern. In other words, a concern is named a fuzzy set [17].

3.2 The ConcernMapper API

In ConcernMapper, the concern model is accessible through an application programming interface (API) that implements the Façade design pattern [3]. The internal implementation of the model is completely hidden and client code need only interact with the `ConcernModel` class. The `ConcernModel` class supports operations for creating, renaming, and deleting concerns, as well as for adding and removing elements to and from concerns and querying various aspects of the model. Finally the API provides the operations necessary to register and deregister objects listening to changes to the model (an implementation of the Observer design pattern [3]). Since modifications to the model can only be done through the `ConcernMapper` class, all operations resulting in a change to the internal state of the model automatically result in a notification to observers. In other words, the notification logic is completely hidden from clients. The complete model API can be accessed from the source code distributed with ConcernMapper.

3.3 Current Instantiation of the model

ConcernMapper release 1.0.0 only supports populating concerns with fields and methods. The decision stems from an initial goal to provide a simple and robust implementation of the model, but also from our extensive experimentation with FEAT, a previously-developed concern modeling tool [11]. Regarding the decision not to model intra-method elements (such as local variables), our experience with FEAT indicated that modeling concerns at this level of detail did not appear to be a cost-effective strategy. Indeed, to mark specific intra-method details as relevant to a concern requires a developer to spend more effort reasoning about these details than necessary. This is particularly true of code segments which, although they help a developer understand a concern, do not need to be modified during a specific task. Regarding the decision not to support the inclusion of classes, our rationale was that the inclusion of a class as part of a concern model is ambiguous: does it indicate that all of the code in the class implements the concern or only part of it? This aspect was modeled explicitly in an early expression of concern models [9] but has since been dropped. In the current version of ConcernMapper, we solve the problem by displaying classes declaring elements that are part of a concern model: it is thus possible to "add" a class to a model by dragging and dropping a selection of all the elements of the class.

4. EXTENDING CONCERNMAPPER

We are offering ConcernMapper as a simple, extensible platform for experimentation with advanced separation of concerns techniques. This section highlights our motivation and demonstrates, through an example, the simplicity of extending ConcernMapper for other research applications.

4.1 Motivation

Building concern models manually is only one of many ways of producing information about the implementation of concerns in source code. Different techniques have recently been proposed by software engineering researchers that could lead to automatic generation of concern models. Such techniques include (but are not limited to):

- **Program Navigation Analysis** [12]. A number of approaches have been proposed to monitor and analyze the actions of developers as they perform software development tasks [5, 10, 13]. The results of such analyses often represent a subset of the code of interest to a developer. This code can be recorded as a concern model.

- **Static Analysis**. Various analyses can be performed on the structural dependencies of programs to elicit code of potential interest. We are currently experimenting with an algorithm to infer code of potential interest to developers based on an analysis of the topology of structural dependencies to a set of interest [7]. Our research prototype for this algorithm is based on ConcernMapper.

- **Feature Location.** A number of techniques can automatically produce an estimate of the methods implementing a feature by analyzing traces of the execution of a system [2, 14]

- **Repository Mining.** Data mining techniques have been proposed that report on elements that are often changed together during program evolution tasks [15, 18]. It may be useful to document such change sets as concern models.

- **Information Retrieval.** Information retrieval techniques can be used to automatically associate text (e.g., from user manuals) with the corresponding code [1, 16].

Finally, by storing a core concern model that can be accessed by other Eclipse plug-ins, ConcernModel also provides a simple platform for experimenting with concern visualization techniques.

4.2 Example

We demonstrate below how easy it is to extend ConcernMapper through a complete example. For our example we have chosen to build a plug-in that creates a concern through a simple static analysis technique. Using the Eclipse search engine, our example plug-in finds all the methods accessing a field selected by the user and adds them to a new concern in the ConcernMapper view. Figure 5 shows how this action is triggered in our plug-in (the pop-up menu has been artificially simplified for clarity in the presentation).

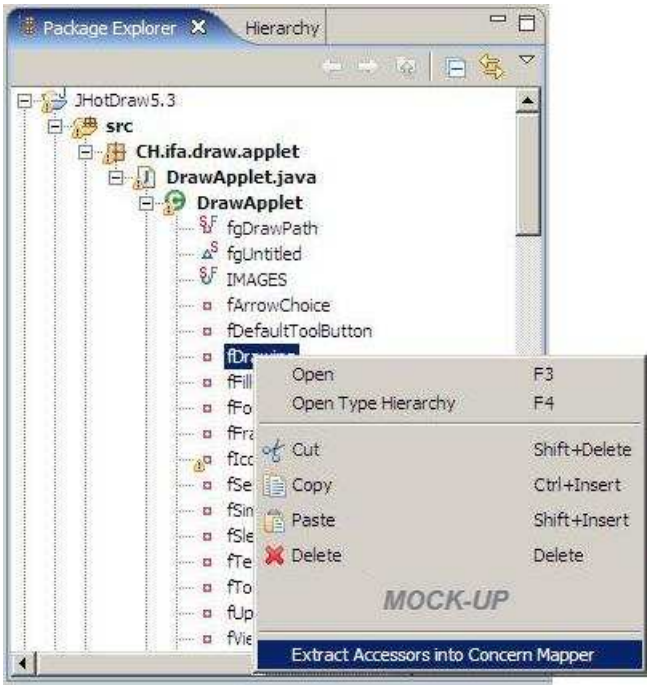

Figure 5: Creating a concern representing field accessors.

We developed this plug-in in three steps.

1. **Building the user interface:** Our example plug-in contributes an action to `IField` elements' popup menu. To do so, we extended the `org.eclipse.ui.popupmenus` extension point by adding an `objectContribution` to the standard popup menu for `IFields`. This enables users to right-click on any field in the Package Explorer (or other JDT views) and be presented with the possibility to extract accessors of the selected field into a new concern (see Figure 5).

2. **Implementing the action:** We wrote a class that implements the `IObjectActionDelegate` interface and that corresponds to the class named in the extension point (see above). In this class, we implemented the `selectionChanged` method to ensure that the selected element is adaptable to an `IField`. Using the Eclipse search engine, our action searches the workspace for accessors of the selected field and returns them as a `Set`.

3. **Updating the concern model:** So far we did not need to reference the ConcernMapper plug-in. This means that developers can use any other technique to find elements of code that could be part of a concern (see Section 4.1). Elements of interest can be added to the concern model by interacting directly with the `ConcernModel` class, which provides a simple API. Any plug-in needing to extend ConcernMapper's functionality must declare `ca.mcgill.cs.serg.cm` as a required plug-in in the plug-in manifest file. Once this is done, the concern model can be accessed by calling `ConcernMapper.getDefault().getConcernModel()`. Methods such as `newConcern(String name)` and `addElement(String name, Object element, int degree)` make creating concerns and adding elements easy since the plug-in views are automatically refreshed when the model changes. Figure 6 shows the code of the method called when the concern generation action is triggered (with the analysis and exception handling constructs elided for clarity). This code, located in the `run` method of the action delegate, performs the following actions.

   (a) Using the Eclipse search engine, obtain the list of methods calling the selected field (represented by the comments on line 3-4).

   (b) Generate a name for the concern (line 7).

   (c) Using the ConcernMapper API, add a new (empty) concern with the generated name (lines 9-11).

   (d) For each accessor method detected, add the method to the concern (lines 13-17).

That's all. The ConcernMapper plug-in takes care of refreshing the view, and showing which elements are now part of the concern model in the various JDT views. The concern model can then be saved as an XML file by the click of a button.

5. RELATED WORK

The ConcernMapper plug-in evolved from first author's work on FEAT [11]. With ConcernMapper, we are investigating ways to simplify the manual creation of concern models, and to facilitate the programmatic generation of concern models by other tools. The main difference between ConcernMapper and tools such as FEAT and the Concern Manipulation Environment (CME) [4] is that ConcernMapper supports a fuzzy, exclusively extensional model for concerns. Such a model allows users (humans or programs) to create concern representations without having to reason about the structure of concerns a priori. This approach differs from FEAT and CME's crisp, relation-based concern models. ConcernMapper's simpler model directly enables us to leaverage features of Eclipse such as the dragging and dropping of Java elements to give developers a fluid way to create concerns that naturally aligns with their program investigation activities (see Section 2). We are currently investigating how to carry over the main benefits of relation-based concern models (e.g., robustness in the face of software evolution) to our fuzzy concern model.

```
1  public void run(IAction action)
2  {
3    // Search for the accessing IMethod objects
4    // and store them in a Set variable lMethods
5
6    //Add a concern to the concern model
7    String lName = "Accessors of " + aField.getElementName();
8
9    if(!ConcernMapper.getDefault().getConcernModel().exists(lName))
10   {
11     ConcernMapper.getDefault().getConcernModel().newConcern(lName);
12     //Add the accessors to the concern
13     for( Iterator i = lMethods.iterator(); i.hasNext(); )
14     {
15       IMethod lNext = (IMethod)i.next();
16       ConcernMapper.getDefault().getConcernModel().addElement(lName, lNext, 100);
17     }
18   }
19 }
```

Figure 6: Code required to update ConcernMapper

6. CONCLUSIONS

Concerns are not often perfectly separated. As a result, developers often have to perform change tasks involving code scattered through different modules (classes, methods, etc.). Although many features of Eclipse greatly facilitate navigating between different code locations of interest to a developer, the non-localization of code relevant to a concern implies that developers' effort might be wasted tracking down code that is conceptually related and that should be co-located, at least in the context of a specific change task.

In our effort to mitigate this problem we developed ConcernMapper, a simple Eclipse plug-in that supports a concern-oriented approach to software development by allowing developers to quickly build models of the code of interest, and to store these concern models for future use.

ConcernMapper is intended to provide a basic building block for research on separation of concerns, automatic feature location, and program navigation. As such, it follows a simple architecture that is easy to extend.

7. ACKNOWLEDGMENTS

The authors are grateful to the anonymous reviewers for their useful comments. This work is supported by an Eclipse Innovation Award, a research grant from the Natural Sciences and Engineering Research Council of Canada (NSERC), and by the Faculty of Science of McGill University.

8. REFERENCES

[1] 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, 1999.

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

[3] 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] William Harrison, Harold Ossher, Stanley Sutton Jr., and Peri Tarr. Concern modeling in the Concern Manipulation Environment. In *Proceedings of the First International Workshop on the Modeling and Analysis of Concerns in Software (MACS)*, volume 30 (4) of *ACM SIGSOFT Software Engineering Notes*, 2005.

[5] 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.

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

[7] Martin P. Robillard. Automatic generation of suggestions for program investigation. In *Proceedings of the Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 11–20, 2005.

[8] Martin P. Robillard, Wesley Coelho, and Gail C. Murphy. How effective developers investigate source code: An exploratory study. *IEEE Transactions on Software Engineering*, 30(12):889–903, 2004.

[9] 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.

[10] 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.

[11] 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, 2003.

[12] 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.

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

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

[15] Annie T.T. Ying, Gail C. Murphy, Raymond Ng, and Mark C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9):574–586, 2004.

[16] Wei Zhao, Lu Zhang, Yin Liu, Jiasu Sun, and Fuqing Yang. SNIAFL: Towards a static non-interactive approach to feature location. In *Proceedings of the 26th International Conference on Software Engineering*, pages 293–303, 2004.

[17] H.-J. Zimmermann. *Fuzzy Set Theory and Its Applications*. Kluwer Academic Publishers, third edition, 1996.

[18] 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.