In *Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pp. 479–482.
IEEE Computer Society Press, September 2006. © IEEE, 2006.

Tracking Concerns in Evolving Source Code: An Empirical Study

Martin P. Robillard
School of Computer Science
McGill University
Montréal, QC, Canada
martin@cs.mcgill.ca

Abstract

The association between the description of a concern (e.g., a feature) and the code that implements it is valuable information that can degrade as the code of a system evolves. We present a study of the evolution of the implementation of a concern in 33 versions of an open-source text editor. We represented the implementation of the concern using concern graphs, a model that was designed to be resilient to source code evolution. The study showed how the concern graph model supports tracking a concern's implementation in an evolving system, as well as inferring high-level past changes and assessing the stability of the concern's implementation.

1. Introduction

Source code modifications often address concerns, or features, whose implementation is scattered across a number of modules. In such cases, developers often have to spend a significant amount of effort investigating the system to identify all the code locations which may be associated with the change. When repeated changes address a same scattered concern, the continual re-investigation of the code can translate into inefficiencies of the software development process.

One way to mitigate this problem is to use annotations or artifacts to document how various parts of the source code relate to different concerns. A variety of tools can help developers view and navigate this knowledge to ease software development tasks. One challenge with this approach, however, is that every time a system is modified, the concern documentation is at risk of becoming invalid.

We are currently developing and evaluating ways to model concerns that can withstand the destructive effects of source code evolution. In one of our approaches, *concern graphs* [7], we represent the implementation of a concern in a way that makes it possible to automatically detect when the description of a concern is no longer consistent with the code, and to provide support for automatically or semi-automatically updating the concern description.

To investigate the practical benefits of this technique, we studied the modifications to a feature over 33 versions of an open-source text editor, spanning over four years of development history. Overall, the study showed *how* concern graphs enable the tracking of concern code over a changing implementation. The study also showed that by analyzing how concern representations become inconsistent over time, we can potentially infer the high-level nature of past changes to the code, and assess the stability of a concern's implementation.

The contributions of this paper include an experimental design for studying how the implementation of a concern evolves using concern graphs, a historical study of a concern's evolution that can serve as a benchmark for assessing other concern modeling techniques, and a series of proposed heuristics for automatically inferring high-level past changes to source code.

2. Background

A concern graph is an artifact that represents the implementation of a concern in source code by documenting the relations between the different program elements involved in the concern's implementation (fields, methods, etc.).[1]

In the concern graph framework, a *concern* is a named collection of *fragments*. A fragment represents a basic relation between program elements that are relevant to a concern's implementation. The definition of a fragment includes an *intension* and its corresponding *extension*. We define *intension* as a structural query involving a program element. For example "`C.f accessed by ALL`" is an intension stating that we are interested in all the methods accessing field `C.f`. In contrast, an *extension* is an exhaustively-specified list of elements (e.g., `C.m1()`, `C.m2(),...`). In a fragment, the extension consists of the

[1]For the purpose of this paper, we present a *simplified* version of the concern graph framework. For additional details, see an extended overview [6] and the complete report [7].

list of elements corresponding to the range of the intension. In our example, this would mean all the methods *actually* accessing field `C.f` in a specific version of a system. In some cases, a fragment must capture a relation between two specific elements (e.g., `A.m1() calls B.m2()`). In this case the "intension" actually includes a range specification (`B.m2()`), and the extension is simply that range element (`B.m2()`). However, by experience, most fragments tend to involve an intension with a free (unspecified) range. We call such fragments *intensional fragments*. The set of *participants* for a concern is the set of all elements found in fragment intensions and extensions.

By combining an intension and its extension in a fragment, whenever the program evolves, the intension can be projected onto new versions of the program to determine if the *generated* extension still corresponds to the *stored* extension. Inconsistencies between the generated and the stored extensions indicate modifications that invalidate the concern graph (a concern graph inconsistency). To repair a concern graph inconsistency, we can update the fragment by replacing the stored extension with the generated extension. We call this operation *synchronizing* the fragment.

In practice, concern graphs are created and used with an Eclipse[2] plug-in called FEAT [7]. FEAT augments Eclipse with a number of search facilities for program investigation (e.g., to obtain all the accessors of a field) that allow a user to add the entire results of a search as a fragment in a concern (the intension is the query and its extension is the query results). Every time source code in an Eclipse project associated with a concern is modified (or when a concern graph is loaded), FEAT re-projects the intension of each fragment and checks the resulting extension for inconsistencies with the stored extension.

3. A Study of jEdit

We studied the potential benefits of combining intensions and extensions to track concerns in evolving source code by conducting an exploratory analysis of the development history of an open-source project.

3.1. Study Questions

For this study, we were interested in documenting a case of concern evolution to use as a benchmark for studies of software maintenance, and to answer the following questions: *a)* How does the combination of intensions and extensions help us track the code implementing a concern?, and *b)* What else can the analysis of intensions and extensions tell us about the evolution of a concern's implementation?

[2]www.eclipse.org

3.2. Target System and Concern

As our target system we chose the jEdit text editor.[3] jEdit is developed in Java and comprises between 60 and 92 kLOC (depending on the version considered). It is a mature project whose history spans over four years.

The concern we chose to investigate is a feature allowing users to "mark" lines in a file. Using items in a top-level "Markers" menu or shortcut keys, jEdit users can add and remove markers, jump to markers, set the marker color, etc. The implementation of this feature is scattered throughout at least eight Java files. We will refer to this feature and its implementation as the MARKERS concern.

3.3. Methodology

We analyzed the evolution of the MARKERS concern in jEdit by creating a concern graph representing the implementation of MARKERS in an early version of the system, and by sequentially loading and adapting the concern graph on each subsequent tagged version of jEdit available in its public CVS repository.

To create our initial concern graph, we checked out version 4.0-pre1 from the jEdit CVS repository into an Eclipse project. We then performed a regular expression search on the term "marker". From the results, we culled the false positives and used FEAT to explore program dependencies to and from the elements found. During the exploration of the code with FEAT, we built a concern graph representing the MARKERS concern.

The initial concern graph comprises 35 fragments, of which 24 are intensional. For example, the intensional fragment `Buffer.markers accessed by ALL` specifies that all the methods accessing field `markers` of class `Buffer` are involved in the implementation of the MARKERS concern.

For each of the 35 fragments, the extension is generated automatically by FEAT and appended to the concern graph definition. In total, the MARKERS concern graph for version 4.0-pre1 involves 60 participants defined in 11 different classes.

To complete the study, we checked out subsequent versions of jEdit in chronological order and analyzed, for each, the impact of the changes on the MARKERS concern. Specifically, given version v_n, its successor v_{n+1}, and a concern graph C_n valid for version v_n, we:

1. Recorded the number of participants in C_n whose code had changed between v_n and v_{n+1}. We obtained this number by comparing each compilation unit in v_{n+1} declaring a participant in C_n and analyzing whether there were code changes in the lexical scope of the participant. We refer to these units of change as *changed participants*.

[3]www.jedit.org

2. Loaded C_n on version v_{n+1} and recorded each fragment whose extension was inconsistent between v_n and v_{n+1}.
3. Repaired C_n by synchronizing fragments, removing invalid fragments, and adding new fragments corresponding to new MARKERS code, as appropriate. This last step results in the creation of a new concern graph C_{n+1}, which we used for the following version.

We followed this method on the 33 non-identical versions of jEdit available, from version 4.0-pre1 (November 2001) to version 4.3-pre3 (1 January 2006). As a result we were able to analyze 32 version differences.

3.4. Quantitative Results

Figure 1 presents a synthesis of the evolution of jEdit and of the MARKERS concern during the period covered by our study. On the figure, the line plots the size of the total jEdit source code in LOC (right vertical axis) as a function of time (horizontal axis). On the size graph, each diamond represents a jEdit version. As the graph shows, jEdit has significantly grown in the period covered by our study, going from 60 kLOC in 2001 to 92kLOC in the last version (4.3-pre3, 1 January 2006).

In addition to general information about the evolution of jEdit, Figure 1 includes information specific to the evolution of the MARKERS concern. Below each diamond, a column indicates the number of changed participants (with respect to the previous version, left axis). For example, the third bar from the left corresponds to version 4.0-pre4. Its height of 14 indicates that between version 4.0-pre4 and the previous version (4.0-pre3) there were 14 changed participants. In general, analysis of the number of changed participants shows that almost every version of jEdit contains some changes to fields or methods involved in the implementation of MARKERS. Finally, the diameter of the circles in Figure 1 represents, for each version, the number of inconsistent fragments detected when loading the concern graph on the code of this version. The numbers vary from 2 to 21.

In our case, only 10 of the 32 version differences studied caused inconsistencies to appear in the concern graph. Such a low ratio is expected as concern graphs only capture the essential structure of concerns as can be represented by non-local relations between fields, methods, and classes. As such, any modification that does not affect the relations captured by a concern graph will not alter the quality of the information that it represents. In the case where modifications result in observable changes at the level of concern graphs, we found that the inconsistencies detected could be very useful in reasoning about the implementation of the concern.

3.5. Qualitative Analysis

For a majority of versions differences, we were able to associate a group of inconsistencies with a single, high-level change. When possible, we documented our observations as heuristics for inferring the nature of a change. We describe four illustrative heuristics.

In the following descriptions, we refer to the extension of a fragment as stored in the concern graph as the *stored extension* (E_s) and the extension generated by projecting the intension on the source code as the *generated extension* (E_g). An inconsistency is detected whenever $E_g \neq E_s$. For an inconsistent fragment, an element e can be either *extraneous* ($e \notin E_s \wedge e \in E_g$), *missing* ($e \in E_s \wedge e \notin E_g$), or *invalid* ($e \in E_s \wedge e$ does not exists in the program version used to generate E_g).[4]

Heuristic 1 (Element Move) *For a fragment: if an element e of class C is missing and an element e' of class C' is extraneous, and e and e' have the same name, then e was probably moved to class C'.*

Heuristic 2 (Element Rename) *For a set of fragments F: if, $\forall f \in F$, an element e is invalid and an element e' is extraneous, e was probably renamed to e'. Confidence in this heuristic increases with the cardinality of F.*

Heuristic 3 (Code Block Move) *For a set of fragments F: if, $\forall f \in F$, a method m is missing but valid and a method m' is extraneous, a code block was probably moved from m to m'. Confidence in this heuristic increases with the cardinality of F.*

Heuristic 4 (Pull Down Method) *For a fragment: if an element e of class C is missing and an element e' of class C' is extraneous, and e and e' have the same name, and C' is a subclass of C, then e was probably pulled down into class C'.*

3.6. Concern Change Assessment

Analyzing concern graph inconsistencies can also help assess the relative stability of different parts of the code relating to a concern. One way to assess the (in)stability of a concern's implementation is to count the number of times a given fragment became inconsistent as the results of the evolution of a system.

In our study, analyzing this data allowed us to draw two conclusions. First, a very stable class can be used by very unstable code. For example, one class in jEdit went through only six revisions but the two most unstable fragments of MARKERS were defined on members of that class.

Our second conclusion is that users of aspect-oriented programming languages should be careful when specifying

[4]Note that given e, E_s, E_g, *invalid* $\rightarrow$ *missing*. However, the additional qualification can be useful.

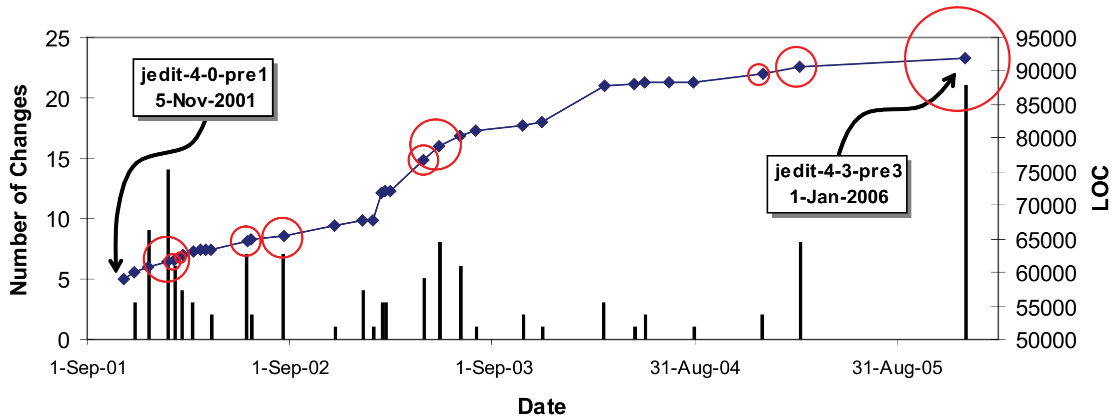

Figure 1. The evolution of the MARKERS concern in jEdit.

pointcuts intensionally as our study produced evidence that the extension corresponding to an pointcut-like intension may be in a constant state of flux. For example, one method of MARKERS went through nine different sets of callers through the history of jEdit.

4. Related Work

A number of approaches have been proposed that allow developers to specify a subset of the source code of a program using different mechanisms based on intensional specifications (e.g., [1, 3, 5]). By contributing this empirical study, we provide a case of concern evolution that can be reused to facilitate the evaluation of research on concern tracking and evolution.

A number of approaches have also been proposed for inferring past refactorings or high-level changes from a system's change history (e.g., [2, 4, 8]). These approaches involve different heuristics that are based on input data that is different from the one we use (concern graphs). However, such research can also benefit from standard evaluation scenarios that are based on empirical data, such as the one provided in this paper.

5. Conclusions

We presented a study of a concern's evolution over four years of development history. The study explored the relation between textual differences in the source code of a concern and inconsistencies in high-level models representing this code (concern graphs). The study provided evidence that combining intensional source code descriptions with their corresponding extensions can support automated reasoning about the concern's evolution and facilitate tracking the implementation of a concern, as well as assessing its stability.

Based on our current support for inconsistency analysis in FEAT, we are currently designing a way to automatically encode, detect, and execute the heuristics presented in this paper. We hope to increase the level of automation with which we can track concern code in evolving software, with the long-term goal of simplifying the maintenance of concerns or features whose implementation is scattered throughout the system.

Acknowledgments

Imran Majid and Ashar Aziz first identified the markers feature as an interesting concern in jEdit. Thanks to Brian de Alwis, Jean-Sébastien Boulanger, Gail Murphy, and the anonymous reviewers for comments on this paper. This work was supported by NSERC.

References

[1] M. Chu-Carroll, J. Wright, and D. Shields. Supporting aggregation in fine grained software configuration management. In *Proc. Int'l Symp. Foundations of Software Engineering*, pages 99–108, 2002.

[2] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *Proc. Conf. Object-oriented Programming, Systems, and Applications*, pages 166–177, 2000.

[3] W. Harrison, H. Ossher, S. Sutton Jr., and P. Tarr. Concern modeling in the concern manipulation environment. Technical Report RC23344, IBM Research, 2004.

[4] J. I. Maletic and M. L. Collard. Supporting source code difference analysis. In *Proc. 20th Int'l Conf. Software Maintenance*, pages 210–219, 2004.

[5] K. Mens, T. Mens, and M. Wermelinger. Maintaining software through intentional source-code views. In *Proc. 14th Int'l Conf. Software Engineering and Knowledge Engineering*, pages 289–296, 2002.

[6] M. P. Robillard. Tracking and assessing the evolution of scattered concerns. In *Proc. AOSD Workshop on Linking Aspect Technology and Evolution*, 2006.

[7] M. P. Robillard and G. C. Murphy. Representing concerns in source code. *ACM Transactions on Software Engineering and Methodology*, 2006. To Appear.

[8] Z. Xing and E. Stroulia. Recognizing refactoring from change tree. In *Proc. 1st Int'l Workshop Refactoring*, pages 41–44, 2003.