

Tracking and Assessing the Evolution of Scattered Concerns

Martin P. Robillard

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

ABSTRACT

In this position paper, we describe how we document the implementation of scattered concerns by combining intensional specifications of relations between program elements and their corresponding extensions for a specific version of a program. We show that this strategy allows us to automatically track the source code implementing a concern as it evolves and to assess the stability of a concern's implementation. We illustrate these benefits with results obtained from ongoing empirical studies of the evolution of scattered concerns.

1. INTRODUCTION

Software 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 a system to identify all the code locations which may be associated with the change. When repeated changes address a same scattered concern, the perpetual re-investigation of the code can directly translate into inefficiencies of the software development process.

One way to address this problem is to explicitly link the description of a concern with the code implementing the concern. With this approach, first proposed by Soloway et al. in 1988 [16], annotations or artifacts document how specific parts of the source code relate to different scattered concerns. Different forms of tool support can help developers view and navigate this information to ease software development tasks. One universal challenge with this approach, however, is that every time a system is modified, the concern documentation is at risk of becoming invalid, and must be constantly maintained.

In our current work on concern representation, we are developing and evaluating ways to model concerns that can withstand the destructive effects of source code evolution. In one of our approach, *concern graphs* [12, 13], we represent the implementation of a concern using both an intensional specification (e.g., “all the callers of method `m1()`”) and the corresponding extension on a specific version of a code base (e.g., `m2()` and `m3()`). This way, it is possible to automatically detect when the code evolves to a point where the projection of an intension on a code base does not correspond to the extension (what we call a concern graph inconsistency).

Past [15] and ongoing legacy [18] studies of concern evolution have provided encouraging evidence that concern graphs can be used to track and assess the evolution of scattered concerns. In the rest of this paper, we describe several observations we made about

the value of combining intensional and extensional specifications to describe the implementation of concerns. Among others, simple heuristics applied to concern graph inconsistencies enable the automatic detection of new concern methods, of changes to method signatures, of class and method moves, of moves to code blocks, etc. In addition, we found that studying the history of concern graph inconsistencies between different versions of a system allowed us to assess the relative stability of a concern's implementation for the purpose of refactoring the concern's code to a different object- or aspect-oriented modularity.

2. BACKGROUND

A concern graph [13] is an artifact representing 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.). In this paper, we present a simplified version of concern graphs that includes only the concepts necessary to understand the paper. The complete description of the most recent version of the concern graph framework can be found in a separate report [12].

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*¹ and its corresponding extension representing the actual range of the relation for a specific version of a program.

For example, if a developer decides that all the accessors of a field `f` in a class `C` are associated with the implementation of a concern, then the following intension is recorded:

```
C.f accessed by ALL.
```

When this intension is recorded, a tool analyzes the current version of the code of the program and determines the extension (e.g., methods `C.m1()` and `C.m2()`). The complete fragment recorded in the concern graph then consists of the intension `C.f accessed by ALL` and its extension `{C.m1(), C.m2()}`.

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).

¹We use the term “intension” in the sense of Eden and Kazman, to denote a structure that can “range over an unbounded domain” [4, p. 150]

In practice, concern graphs are created and used with an Eclipse² plug-in called FEAT [14]. 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. FEAT reports inconsistencies in a specialized Eclipse view that describes which fragments are inconsistent and why they are inconsistent. For example, Figure 1 shows the FEAT Inconsistencies View loaded displaying the details of the inconsistencies for a fragment with the intension

`Marker.createPosition()` called by ALL.

The adornments on the elements in the detailed view (bottom panel) show that the stored extension is missing a call from `parseBufferLocalProperties()` but includes a call from a method that no longer exists, `parseBufferLocalProperties(String)`.

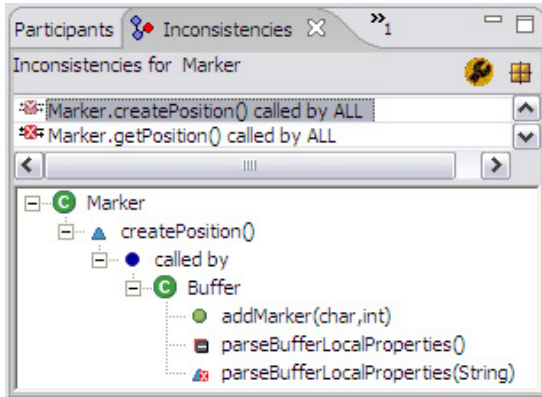


Figure 1: The FEAT Inconsistencies View

3. CONCERN TRACKING

Concern graphs are models that abstract the implementation of a concern as a relation between program elements. As for any model, concern graphs are intended to represent the essential properties of a phenomenon while abstracting away the details. In the case of concern graphs, the essential properties of a concern’s implementation are the inter-procedural relations between program elements (e.g., fields, methods), and the details include the syntax of statements, variable names, local interactions, and comments.

3.1 Minor Code Changes

Details of the code and local interactions have no impact on the information stored in a concern graph. As an example, let us consider the following Java source code

```
class A {
    void m(boolean p) {
        // p is true in the morning
        if (p) x();
        else y();
    }
}
```

²www.eclipse.org

If a developer decides that all of the methods called by `m` are involved in the implementation of a concern, this fact can be specified with the intension:

```
A.m() calls ALL.
```

In a concern graph this intension would be stored in a fragment that would include the extension $\{x(), y()\}$.

In this case, any change that would not involve either removing or adding a method call in `m` would have no effect on the concern graph. For example in the revised version

```
class A {
    void m(boolean p) {
        // p is true at night
        if (!p) x();
        else y();
    }
}
```

the extension corresponding to `m(boolean) calls ALL` is preserved (and the concern graph is likely to remain valid) despite the changes to the comment and branching predicate.

This basic tolerance to changes in source code is an important property of concern graphs that facilitates the tracking of concern code. Since concern graphs are not intended to describe the *behavior* of a concern’s implementation but rather its *location*, the risk of transparently invalidating a concern graph by changing the local implementation of a method is very low. In our case, this would only occur in the case where changes to the source code invalidate the intension while preserving the extension. In other words, we would have to change `x()` or `y()` in a way that renders the method irrelevant to the concern, while preserving all existing relations encoded in fragments. We have not yet encountered such a case in practice.

3.2 Major Code Changes

In addition to minor modifications, a software system will also undergo more important changes that will affect the validity of the concern graph. For example, if we now change our example code fragment to:

```
class A {
    void m(boolean p) {
        // p is true at night
        if (!p) x();
        // No else branch
    }
}
```

the extension corresponding to `m(boolean) calls ALL` is now $\{x()\}$, which is inconsistent with the previous extension $\{x(), y()\}$. Experience and studies of concern graph evolution have demonstrated that analyzing the details of concern graph inconsistencies can give us a wealth of information about the nature of the source code changes.

We are currently studying how concern graph inconsistencies can be translated into high-level information about concern changes that can be used to automatically adapt the concern graph to reflect the code changes. We present a number of our results here in the form of heuristics. In the following descriptions, we use the expression *synchronizing a fragment* to indicate updating a fragment’s stored extension with a generated version.

HEURISTIC 1 (SIGNATURE CHANGE). *If a generated extension is missing a method and contains a spurious method of the same name defined in the same class but with a different parameter list, the missing and spurious elements probably represent a change in the method signature that does not otherwise impact the concern graph. The concern graph can be automatically repaired by synchronizing the inconsistent fragment.*

Figure 1 shows a case where this heuristic applies.

HEURISTIC 2 (MOVE METHOD). *If a generated extension is missing a reference from an element in class C and contains spurious references from an element with the same name in a class C' , the element was probably moved from C to C' . The concern graph can be automatically repaired by synchronizing the inconsistent fragment.*

HEURISTIC 3 (NEW CONCERN ELEMENT). *If a generated extension is missing an element that cannot be associated with any other heuristic, the missing element is probably a new element introduced in the concern. The new element should be inspected by a developer.*

HEURISTIC 4 (METHOD RENAME). *If a reference is missing from a number of extensions and, in each case, there exists a corresponding spurious reference from a method that does not exist, the referring method was probably renamed. The concern graph can be automatically repaired by synchronizing the inconsistent fragment.*

HEURISTIC 5 (CODE BLOCK MOVE). *If a reference is missing from a number of extensions and, in each case, there is a corresponding spurious reference from a method that exists, a concern-related code block may have been moved. Confidence that this heuristic applies increases if a number of fragments present the same inconsistency. The new element should be inspected by a developer.*

Our studies also showed cases where we could refine the Move Method heuristic into the more specific Pull Down Method.

HEURISTIC 6 (PULL DOWN METHOD). *If a generated extension is missing a reference from an element in class C and contains spurious references from an element with the same name in a class C' , and C' is a subclass of C , we can say that the element was probably pulled down from C to C' .*

We are currently designing a way to automatically encode, detect, and execute these heuristics in the FEAT tool, to increase the level of automation with which we can track concern code in evolving software.

4. CONCERN CHANGE ASSESSMENT

In addition to facilitating the tracking of concern-related code throughout the evolution of a system, analyzing concern graph inconsistencies can help assess the relative stability of different parts of the code relating to a concern. This is important when trying to decide whether and how a concern can be refactored, either using traditional object-oriented refactoring [6] aspect-oriented techniques[5].

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 empirical studies, analysis of this factor allowed us to make two simple but important observations. First, a very stable class can be used by very unstable code. In one case, a class that was part of the core implementation of a concern went through only 6 revisions throughout the 4-year history of the system. However, the code referring to this class was in a constant flux that resulted in many concern graph inconsistencies. This example shows that we cannot use naive metrics such as the number of file revisions to assess the stability of a concern's implementation.

Our second observation is that AspectJ [9] programmers should be careful when specifying pointcuts intensionally. We now have a collection of cases showing that, in practice, the extension corresponding to an pointcut-like intension may be changing over time. For example, an AspectJ pointcut may have many different sets of shadows in the history of the system without programmers necessarily being aware of the situation. Although in ideal AOP programs the differences should not matter, in reality they might. Storing the extensions corresponding to intensional specifications and analyzing the inconsistencies between extensions in different versions of a system enables the automatic detection and notification of such cases.

5. RELATED WORK

A number of approaches have been proposed that allow developers to specify a subset of the source code of a program using intensional specifications.

For example, the Stellation [1, 2] software configuration management system supports virtual source files using a typed aggregation mechanism that can intensionally collect different program elements and other artifacts in a single modular unit. The Aspect Browser is a tool proposed to help developers find concerns using lexical searches of the program text [7]. In AspectBrowser, a concern description is intensionally defined as a set of regular expressions. As another example, Intentional Views [10, 11] allow developers to specify different views of a system that reflect some form of commonality, which can include relevance to the implementation of a concern. Intentional Views are specified explicitly using a declarative programming language and are generative (so only the intension is stored). Finally, The Concern Manipulation Environment (CME) [8] includes a Concern Explorer tool that can be used to describe concerns using queries (i.e., intensions) in a way that is similar to FEAT, but without the support for inconsistency analysis between different extensions corresponding to a single intension.

We see the number and variety of approaches for intensional specification of concern code as an exciting indicator that such an idea is feasible and practical. The main difference between concern graphs and previous approaches is that concern graphs store the extensions that correspond to programmer-defined intensions. Only with this property can the analyses described in the paper be possible.

A number of approaches have also been proposed for inferring past refactorings from a system's change history. For example, Demeyer et al. [3] analyze changes in object-oriented metrics (e.g., the number of messages sent) to infer potential past refactorings. Xing and Stroulia [17] developed an approach to recognize past refactorings based on changes in a class hierarchy as documented in an object-oriented design model. In contrast, our inferences are done using concern graphs, an abstraction that is closer to the source code than both metrics and class diagrams.

6. CONCLUSIONS

Artifacts that describe the source code implementing scattered concerns can be helpful to developers performing program evolution tasks. However, an artifact referring to source code typically becomes inconsistent as the code evolves, reducing its effectiveness. In this paper, we described how we can efficiently evolve descriptions of concerns in source code together with the evolution of a system. The key to enabling the inexpensive evolution of concern descriptions is to combine intensional specifications of the concern code with their corresponding extensions. Analysis of the inconsistencies between extensions stored in a concern description and extensions generated on a program by projecting the corresponding intensions can help us determine how to adapt the concern descriptions, and provides us with valuable insights on the stability of a concern's implementation as the code evolve.

Acknowledgments

The author is grateful to Gail Murphy for useful comments on an earlier draft of this paper.

7. REFERENCES

- [1] Mark Chu-Carroll, 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, 2002.
- [2] 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, 2000.
- [3] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. Finding refactorings via change metrics. In *Proceedings of the Conference on Object-Oriented Programming, Systems, and Applications*, pages 166–177, 2000.
- [4] Ammon H. Eden and Rick Kazman. Architecture, design, implementation. In *Proceedings of the 25th International Conference on Software engineering*, pages 149–159, 2003.
- [5] Tzilla elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming. *Communications of the ACM*, 44(10):29–32, 2001.
- [6] Martin Fowler. *Refactoring—Improving the Design of Existing Code*. Object Technologies Series. Addison-Wesley, 2000.
- [7] 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, 2001.
- [8] William Harrison, Harold Ossher, Stanley Sutton Jr., and Peri Tarr. Concern modeling in the concern manipulation environment. Technical Report RC23344, IBM Research, 2004.
- [9] 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, 2001.
- [10] Kim Mens, Tom Mens, and Michel Wermelinger. Maintaining software through intentional source-code views. In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*, pages 289–296, 2002.
- [11] Kim Mens, Bernard Poll, and Sebastián González. Using intentional source-code views to aid software maintenance. In *Proceedings of the International Conference on Software Maintenance*, pages 169–178, 2003.
- [12] Martin P. Robillard. *Representing Concerns in Source Code*. PhD thesis, Department of Computer Science, University of British Columbia, Canada, 2003.
- [13] 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.
- [14] 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.
- [15] Martin P. Robillard and Gail C. Murphy. Evolving descriptions of scattered concerns. Technical Report SOCS-TR-2005.1, School of Computer Science, McGill University, 2005.
- [16] 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, 1988.
- [17] Zhenchang Xing and Eleni Stroulia. Recognizing refactoring from change tree. In *Proceedings of the First International Workshop on Refactoring: Achievements, Challenges, and Effects*, pages 41–44, 2003.
- [18] Martin V. Zelkowitz and Dolores R. Wallace. Experimental models for validating technology. *IEEE Computer*, 31(5):23, 1998.