Inferring Structural Patterns for Concern Traceability in Evolving Software

Barthélémy Dagenais
School of Computer Science
McGill University
Montréal, QC, Canada
bart@cs.mcgill.ca

Silvia Breu*
Computer Laboratory
University of Cambridge
Cambridge, UK
silvia.breu@cl.cam.ac.uk

Frédéric Weigand Warr, and Martin P. Robillard
School of Computer Science
McGill University
Montréal, QC, Canada
{fwwarr, martin}@cs.mcgill.ca

ABSTRACT

As part of the evolution of software systems, effort is often invested to discover in what parts of the source code a feature (or other concern) is implemented. Unfortunately, knowledge about a concern's implementation can become invalid as the system evolves. We propose to mitigate this problem by automatically inferring structural patterns among the elements identified as relevant to a concern's implementation. We then document the inferred patterns as rules that can be checked as the source code evolves. Checking whether structural patterns hold across different versions of a system enables the automatic identification of new elements related to a documented concern. We implemented our technique for Java in an Eclipse plug-in called ISIS4J and applied it to a number of concerns. With a case study spanning 34 versions of the development history of an open-source system, we show how our approach supports the tracking of a concern's implementation through modifications such as extensions and refactorings.

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

General Terms: Documentation, Experimentation

Keywords: Concern tracking, software evolution, feature location, intension template

1. INTRODUCTION

Most software systems involve concerns that are not encapsulated in their own module, for a variety of reasons that do not necessarily question the ability of developers. For example, some concerns are inherently difficult to encapsulate [11], while others become scattered as the result of repeated changes [6]. The scattered nature of source code relating to a given concern means that a developer who must perform modifications that pertain to the concern must also spend effort locating and understanding the concern's implementation. Concern location is an important software engineering activity, and many approaches have been proposed to assist it. Examples include the approach of Eisenbarth et al., who propose to associate source code with features[1] based on the analysis of execution traces [7], or SNIAFL, a technique that associates functions with feature descriptions based on information retrieval techniques [24]. The result of the feature location activity is usually a list of program elements (e.g., methods and fields in an object-oriented language) that are associated with the concern of interest.

Depending on the technique used and the concern under investigation, concern location can require an important amount of developer time. Therefore, the results of the location activities should be preserved as part of the project's artifacts for later consultation by developers. Unfortunately, a concern-to-code mapping (or *concern mapping*) is at risk of becoming invalid with every new change recorded in the source code repository. Even trivial changes such as renaming a few methods can potentially destroy hours of reverse-engineering work if the renamed methods cannot easily be rediscovered.

In this paper, we propose a technique for tracking the implementation of concerns throughout multiple versions of a software system by leveraging existing structural patterns that may exist among the elements in a mapping. We built a system, called ISIS4J, to automatically infer a number of structural patterns among program elements based on a predefined set of pattern templates. As a very simple example, let us assume that for a concern C (e.g., the AUTOSAVE feature in a text editor), a concern location activity yields the mapping $\{m_1, m_2, m_3\}$, consisting of three methods that implement AUTOSAVE. If m_1, m_2, and m_3 correspond to all of the methods in the system that access a field f_1, our system will detect this pattern and store a representation of the pattern instance along with the mapping. Once pattern information is available in addition to basic concern mappings, it is possible to check whether the pattern is still valid in later versions of the system. When the pattern is

*This research was conducted while S. Breu was a visitor in the Software Evolution Research Group at McGill.

[1]In this paper, we use the terms *concern* and *feature* interchangeably, even though we consider a feature to be a special case of a concern.

ASE'07, November 4–9, 2007, Atlanta, Georgia, USA.

no longer valid, it is possible to automatically discover what program elements should be included in the mapping to restore the validity of the pattern. In the above example, we could use the detected pattern to automatically be notified of new accessors to f_1 that may now also play a part in the concern's implementation.

We assume that the program elements that implement a high-level concern exhibit enough commonalities to allow us to infer patterns that will help us track the concern's implementation in an evolving code base. We evaluated this assumption by using ISIS4J in an experiment to infer structural patterns on 16 sample concerns and for a study in which we tracked the implementation of a concern over 34 releases of an open-source system. Our experiment showed that although there is rarely a perfect match for a pattern template, in many cases patterns could be inferred through small and automatic modifications to the concern mappings. Our case study validated the usability of ISIS4J, showed how our approach supports the tracking of a concern's implementation through modifications such as extensions and refactorings, and helped us identify the kinds of evolutionary changes that were likely not to be detected by our current prototype. The contributions of this paper include:

a) a technique to automatically infer patterns based on a given mapping and a set of *pattern templates*;

b) the architecture of a complete and usable system for inferring and checking patterns in concern mappings;

c) a body of empirical evidence describing the performance of our approach under controlled conditions.

In the remainder of this paper, we describe our idea of using patterns to describe concerns (Section 2) and our inference system (Section 3). We then present our empirical evaluation (Sections 4 and 5). We conclude with a description of the related work (Section 6) and our final observations on our technique (Section 7).

2. REPRESENTING CONCERNS IN CODE

Concerns whose implementation is scattered are an important issue in software development, and many approaches have been proposed to help developers keep a record of the code associated with a given concern. The idea of explicitly representing the implementation of scattered concerns can be traced back to Soloway et al., who proposed writing cross-referenced textual documentation as annotations to the code [21]. More recent techniques for documenting scattered concerns have focused on representations that allow a higher level of automatic analysis (e.g., the use of regular expressions to specify segments of code related to a concern [9]). Our work on concern modeling builds on the ideas developed into the FEAT [18] and ConcernMapper [20] tools. In this section, we describe the previous work (Section 2.1) and new ideas (Sections 2.2 and 2.3) that form the background for our structural inference system. Section 6 provides a more extensive discussion of the related work.

2.1 Concern Modeling

Essentially, a concern mapping consists of an association between a high-level concern and a set of source code elements. Such a set of elements can be discovered in a number of ways, including through manual inspection of the code or more sophisticated reverse-engineering techniques.

The simplest way to represent a concern's implementation is to record the source code elements that correspond to its implementation. For example, if a PRINTING feature in a text editor is found to be implemented by four methods $m_1 - m_4$, we would simply record the signatures of the four methods as associated with PRINTING. In this case we consider the concern to be represented *extensionally*, and the list of elements associated with the concern is its *extension*. This simple scheme is the one used by the ConcernMapper concern modeling tool [20]. The major advantage of representing concerns extensionally is the simplicity of the scheme: one can associate source code elements with a concern by listing them. In situations where automatic feature location techniques are used, it is thus possible to take the output of the technique as the concern mapping (an extensional representation). On the other hand, the main weakness of the extensional technique is that it is not resilient to evolutionary changes: modifications to the source code (such as refactorings) will easily invalidate the mapping. For example, renamed elements will not be found in later versions of the system.

One can also model concerns *intensionally*.[2] In this case, elements relating to a concern are not listed explicitly but through patterns, called *intensions*, that describe their characteristics. For example, if all of the methods that access field f_1 are involved in the implementation of PRINTING, the intension `all accessors of` f_1 would be recorded. The use of intensions to represent concerns in source code is supported by a number of concern modeling tools, including CME [22], IntensiVE [15], AspectBrowser [9], and FEAT [18]. The advantage of specifying concerns intensionally is that this scheme is more tolerant of evolutionary changes. For instance, in the case described above, if accessors of field f_1 are added or removed as the code evolves, the changes will automatically be reflected in the concern model. In particular, the FEAT tool implements a model of concerns that combines both intensions and extensions, a technique that has been demonstrated to be robust to many types of evolutionary changes [17, 18]. Unfortunately, representing concerns intensionally requires more effort from developers, and intensional descriptions are not produced by feature location techniques. We have thus developed a technique to automatically infer intensions based on a given mapping and a set of *intension templates*.

2.2 Intension Inference

Given a concern mapping, we can posit a number of potential intensions. For instance, given four methods $m_1 - m_4$ and a field f_1, the following situations could occur:

- Methods $m_1 - m_4$ are all the methods that access f_1.
- Methods $m_2 - m_4$ are all the methods that override m_1 (or any combination thereof).
- Methods $m_1 - m_4$ are all the callers of a fifth method m_5 that is not part of the mapping.
- Methods $m_1 - m_4$ and field f_1 constitute all the members of class C.

[2]We use the term "intension" in the sense of Eden and Kazman, to denote a structure that can "range over an unbounded domain" [3, p.150].

All of the above examples can be encoded as intensions. Our system (described in Section 3) is able to automatically detect, encode, and check such intensions.

We currently consider seven kinds of *intension templates* for Java systems: callersOf, calledBy, accessorsOf, accessedBy, overrides, implements, and declaredBy. In the following definitions, e is the intension criterion (a method, field, or class).

- callersOf(e) denotes all methods in the program that call the same method e.
- calledBy(e) denotes all methods in the program that are called by method e.
- accessorsOf(e) denotes all methods in the program that access (read or write) the same field e.
- accessedBy(e) denotes all fields in the program that are accessed (read or written) by the same method e.
- overrides(e) denotes all methods in the program that override a method e.
- implements(e) denotes all methods in the program that implement method e that is declared in an interface or abstract class.
- declaredBy(e) denotes all methods and fields that are declared within the same class e.

Finally, we define the apply operator on intensions. Given an intension $\mathcal{I}(e)$, apply($\mathcal{I}(e)$) produces the extension corresponding to the elements matching the intension on a given version of the program.[3] For example, apply(accessorsOf(f_1)) returns all the methods accessing field f_1.

2.3 Confidence

In practice, it is doubtful that we will find intensions whose extension is completely within a mapping. Therefore, we need a way to express such incomplete intensions within concern mappings. We call the degree with which an intension is matched the *confidence* of an intension.

Given a concern mapping $\mathcal{C}$ and an intension $\mathcal{I}(e)$ with criterion e, we define the confidence of intension $\mathcal{I}(e)$ with respect to $\mathcal{C}$ to be the proportion of concern elements $c \in \mathcal{C}$ that are in the extension of $\mathcal{I}(e)$:

$$\textbf{confidence}(\mathcal{I}(\mathbf{e}), \mathcal{C}) = \frac{|\mathcal{C} \cap \mathsf{apply}(\mathcal{I}(e))|}{|\mathsf{apply}(\mathcal{I}(e))|}$$

For example, if a field f_1 is accessed by four methods m_1–m_4 but only m_1 through m_3 are in concern mapping $\mathcal{C}$, we would say that the confidence of intension accessorsOf(f_1) with respect to $\mathcal{C}$ is 3/4 or 75%.

3. THE ISIS4J SYSTEM

The Implicit Structure Inference System for Java (ISIS4J) is a tool to infer and apply intensions that describe the *implicit structure* of a concern. We define implicit structure as the set of intensions that can be inferred for a concern mapping at a given level of confidence. We implemented ISIS4J as a plug-in for the Eclipse platform.[4] Before presenting the details of the ISIS4J implementation, we describe an example concern and illustrate how ISIS4J can help maintain the knowledge of the concern's implementation as the system evolves. Figure 1 presents an overview of the data-flow through ISIS4J, as tailored for our example.

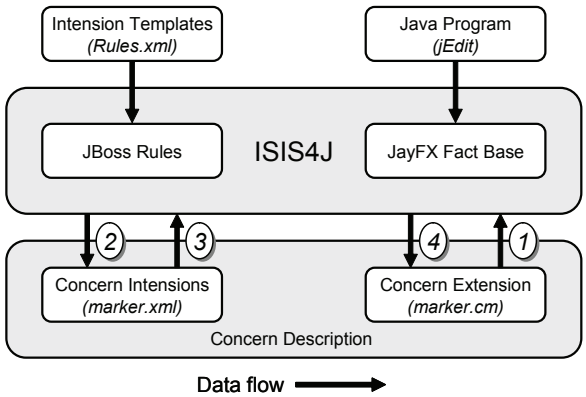

Figure 1: ISIS4J Overview

3.1 Typical Usage Scenario

We base our example on the jEdit open-source text editor.[5] jEdit includes a feature that allows users to "mark" certain lines in a file (the MARKER feature). During a modification to the MARKER feature, a developer identifies a number of methods that are associated with the feature. These could be recorded manually by the developer using a tool like ConcernMapper [20], or simply obtained as the list of changed methods when committing the change. The developer then runs ISIS4J to infer the implicit structure of the mapping for the MARKER feature, producing a collection of intensions (the marker.xml file, Figure 1). This step is represented by arrows 1 and 2 in Figure 1. A number of releases later, the developer needs to perform a second modification task on the same feature. The developer executes ISIS4J, using the marker set of intensions (marker.xml, arrow 3) and the original concern mapping file (marker.cm, arrow 1). ISIS4J removes the elements listed in the concern mapping file that no longer exist (e.g., a method that was deleted) and applies the intensions to see if any new elements are discovered (e.g., a new method accessing a field identified by the accessorsOf intension). ISIS4J then adds the newly discovered elements in the concern mapping file (marker.cm, arrow 4) so the developer can inspect these elements for relevance and proceed with the task.

3.2 Implementation

The above scenario is completely supported by our ISIS4J prototype, described as follows:

Explicit Concern Model. The main input of ISIS4J is a concern mapping file (e.g., marker.cm) created using the ConcernMapper plug-in. In our current implementation, the user creates the initial definition of the concern of interest by explicitly selecting Java methods and fields from a Java project and adding them to the concern mapping by dropping them into the ConcernMapper view.

Data Mining and Rule Engine. Once a concern of interest has been defined, ISIS4J uses the JayFX[6] Eclipse plug-in to create a fact base describing structural relations such

[3]The program version is an implicit parameter of apply that is not mentioned because it will be clear from the context.
[4]www.eclipse.org
[5]www.jedit.org
[6]www.cs.mcgill.ca/~swevo/jayfx/

```
<rule name="inferDeclaredBy">
<!-- All methods declared by the given class are
part of the given concern -->
 <condition>
  <column id="concern" type="Concern"/>
  <column id="class" type="ClassElement"/>
  <eval>
   checkConfidence(Relation.DECLARES,class,concern)
  </eval>
 </condition>
 <consequence>
  RuleWriter.createIntension("declaredBy",
      Relation.DECLARES,class);
 </consequence>
</rule>
```

Figure 2: Example Rule Template

as method calls, inheritance relationships, and field accesses in the source code of the project. For the method call relation, JayFX uses class hierarchy analysis (CHA) [2]. CHA conservatively infers the target of dynamically-dispatched calls by traversing the class hierarchy to discover all applicable bindings. ISIS4J then infers the implicit structure of the concern by checking the elements of a concern mapping against the fact base. ISIS4J performs this data mining operation using the JBoss Rules rule engine[7] and a set of rule templates that we defined in the Drools 3.0 XML language[8] (Rules.xml, Figure 1). Rule templates expressed in Drools correspond to our intension templates as described in Section 2.2. For example, as illustrated by Figure 2, there is a rule template responsible for discovering whether all members of a given class are contained in the user-defined mapping. If such a situation occurs, that rule template would then generate an instance of the declaredBy intension template defined in Section 2.2. In other words, ISIS4J goes through all Java elements in the project (interfaces, classes, methods, and fields) and, for each element, computes the confidence of each one of the seven intension templates with respect to the mapping. If the computed extension meets a predefined confidence threshold, ISIS4J stores an intension associated with this particular element.

Implicit Structure as a Set of Intensions. The output of ISIS4J is an executable rule set describing the inferred intensions and written in the Drools 3.0 XML language (e.g., marker.xml). We chose to use rules to express intensions to facilitate the integration of the rule engine with our system. In theory, intensions could be expressed in a variety of other mechanisms, such as concern graphs [18] or JQuery queries [10]. With intensions encoded as rules, we can run ISIS4J on any version of the Java project to add to the concern mapping all of the Java elements that correspond to its implicit structure. We refer to this step as *completing* the mapping, and additional elements are the result of *intension-based completion* (see Figure 3). Given a mapping $\mathcal{C}$ and an intension $\mathcal{I}(e)$, we define this operation as $\mathsf{expand}(\mathcal{I}(e), \mathcal{C}) = \mathcal{C} \cup \mathsf{apply}(\mathcal{I}(e))$. It should be noted that, in practice, there might exist Java elements in a concern mapping that are not described by any intension.

Customizable Inference. As mentioned in Section 2.3, ISIS4J can infer intensions describing the concern structure even if not all of the elements matched by the intensions are present in the original concern mapping, provided that their confidence reaches a certain threshold. This threshold is expressed as an integer value between 0 and 100 representing the required confidence level in percentage and can be set in the Rules.xml file by the user. ISIS4J uses this parameter when it infers a concern's implicit structure. Allowing a threshold under 100% raises the issue that if ISIS4J completes a mapping by adding missing elements to the concern, the presence of the added elements might increase the confidence for other intensions above the threshold, allowing those intensions to be added to the intension set. We thus introduced an option to run ISIS4J until it reaches a fixed point (where new intensions can no longer be inferred).

[7]www.jboss.com/products/rules

[8]drools.org/drools-3.0

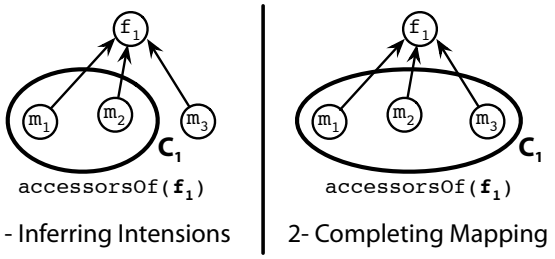

Figure 3: Intension-based Completion of Concern $c1$

System	Version	LOC	Concern Size
GanttProject	2.0.2	43 246	22, 30, 26, 13
Jajuk	1.2	30 679	14, 18, 17, 11
jBidWatcher	1.0pre6	23 051	20, 20, 10, 19
Freemind	0.8.0	70 435	18, 28, 16, 14

Table 1: Characteristics of Target Systems

Runtime Performance. The runtime performance of ISIS4J easily allows its application with every transaction to the source code repository. For example, it took less than one minute to infer and complete a concern mapping on jEdit version 4.0-pre1 (59 kLOC) on a Pentium M 1.8GHz with 1GB of memory.

4. IMPLICIT STRUCTURE

For ISIS4J to be useful to developers requires concern mappings to have latent intensions. We investigated the nature and characteristics of the intensions produced by ISIS4J on a number of concern mappings independently produced by different subjects on four open-source systems.

4.1 Research Questions

We evaluated our assumption that the implementation of concerns has an implicit structure by assessing specific characteristics of our approach:

1. What is the impact of the confidence threshold on the number of intensions detected?
2. ISIS4J can infer intensions in a single pass, or can repeat this process until fixed point is reached and no new intensions can be inferred. What are the tradeoffs of using fixed point inference versus single pass inference (see Section 3.2)?
3. We defined seven intension templates. Are all of these equally useful for generating intensions?

4.2 Experimental Design

To collect initial evidence that would allow us to answer the questions in Section 4.1, we analyzed 16 concerns obtained from a previous empirical study of feature location [19].

In this study, different subjects created concern mappings for 16 different concerns in four different systems (Table 1). Our target systems were all non-trivial, mature projects. As for the concern descriptions, they consisted of a paragraph of text written by two investigators as part of a different study. The concern descriptions were created by explaining a feature of the system as mentioned in the user manual, help pages, or user interface of the system. The complete experimental package associated with this study is available on-line.[9]

For each concern, three different subjects were asked to identify the fields and methods that were judged to be the most relevant to the implementation of the concern. The subjects were undergraduate and graduate students with Java programming experience in Eclipse. No method, process, or tool (besides the features of the Eclipse environment) were given to them to complete this task. This process resulted in $16 \times 3 = 48$ different mappings. Since we were interested in studying the documentation of relatively large sets of elements corresponding to a concern's implementation, we selected, for each concern, the mapping with the highest number of elements. This resulted in 16 data points corresponding to one mapping for each concern (four in each of our target systems). In Table 1, the last column lists the cardinality of each of the four mappings considered for each system. For example, for GanttProject, our four concern mappings included 22, 30, 26, and 13 elements, respectively. The average mapping size across our 16 data points is 18.5 elements. Our 16 sample concern mappings thus represent the fields and methods that would be identified as relevant to a concern as manually determined by a programmer.

For each concern mapping, we used ISIS4J to infer intensions. We then completed the concern mappings based on those intensions (i.e., for incomplete but above-threshold intensions, we added the elements missing in the intensions' extension). We automated ISIS4J and parameterized its execution according to two variables.

Fixed Point. We evaluated the impact of reaching a fixed point where no more new intensions could be inferred versus inferring intensions only once in the initial concern mapping.

Threshold. We tested the effect of four different confidence thresholds: 60%, 75%, 90% and 100%.

4.3 Results

Table 2 shows our results. For each concern (Cn.), for different thresholds, with both the single pass (S) and fixed point (FP) inference options, the table lists the number of intensions detected with ISIS4J and the number of elements added through intension-based completion (in parentheses). The last column of the table provides the average number of detected intensions of each row (Avg.). The original concern mappings size is reported in Table 1 (mappings 1-4 for GanttProject, 5-8 for Jajuk, 9-12 for jBidWatcher and 13-16 for Freemind). For example, for concern 1, with a threshold of 75% and using single pass inference, ISIS4J inferred two intensions, and completing the mapping based on these intensions resulted in adding three elements to the original concern mapping's 22 elements (see Table 2). Based on this data, we provide answers to each of our research questions.

Impact of Confidence on Number of Intensions. For a threshold above 90%, almost no intensions were inferred.[10] We see that using a more relaxed threshold (of 60%–75%) yields, in many cases, the inference of intensions among the elements of a concern. This observation highlights the fact that in order to preserve intensional knowledge about the implementation of a concern, the extension corresponding to latent intensions must often be completed.

Fixed Point Inference Tradeoffs. Fixed point inference only made a difference when using a threshold of 60%. ISIS4J also inferred an extreme number of intensions using fixed point for concern mapping 3 for a threshold of 60%. Completing the mapping based on these intensions resulted in the addition of 75 new elements. Although rare, this situation illustrates the potential cascading effect of fixed point at a low threshold. Since aggregated values based on the data of Table 2 are sensitive to extreme values, we considered concern 3 an outlier and did not take it into account in our calculation of overall quantitative assessments.

With fixed point inference, over all 15 concerns, ISIS4J discovered 52% more intensions than with single pass inference, for a total of 100 intensions instead of 66 (the average is respectively 6.6 and 4.4 intensions per concern mapping). A higher number of intensions per concern mapping is desirable when tracking a concern in an evolving system since ISIS4J can then track more types of changes.

On average, intension-based completion performed with fixed point inference made the size of the concern mappings grow by 24% while single pass inference made it grow by only 14%. In both cases, the size of 75% of all concern mappings did not grow by more than 30%. Thus, most of the final mappings produced by ISIS4J were representative of the original mapping done by the subject. This provides evidence that the quality and relevance of concern mappings produced by ISIS4J do not degrade significantly.

For both fixed point and single pass inference, intensions described between 44% and 86% of the elements in most of the concern mappings (11 out of 15). This suggests that ISIS4J is capable of inferring an implicit structure representative of most of the concern mapping.

Although fixed point inference produced 52% more intensions, these intensions described no more than 7% of additional elements. This observation indicates that the additional intensions mostly described elements that were already in the extension of existing intensions. For example, if we take method m_1 that accesses fields f_1 and f_2, a single pass inference might detect the $\mathsf{accessorsOf}(f_1)$ intension while the fixed point inference will detect an additional $\mathsf{accessorsOf}(f_2)$ intension: both intensions describe the same element, m_1. Redundant intensions can be useful to track changes as they are potentially more robust: in our example, even if field f_1 is refactored or deleted, we can still track the accessors of f_2.

We conclude that performing fixed point inference is desirable as it makes the concern's implicit structure potentially more robust to changes while not altering significantly the nature of the original concern mapping.

Use of Intension Templates. In all of our sample concerns, neither an overrides intension nor an implements intension was inferred. The number of declaredBy as well as accessedBy intensions inferred varies between 0 and 2 per concern. The distribution of the remaining three inten-

[9] http://www.cs.mcgill.ca/~martin/concerns

[10] Results for 90% and 100% thresholds are identical.

	Cn.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	Avg.
60%	**S**	16(10)	3(3)	5(5)	3(1)	0(0)	8(4)	2(1)	2(2)	2(3)	5(5)	0(0)	3(1)	4(4)	12(8)	2(1)	4(0)	4.4
	FP	25(22)	11(12)	83(75)	3(1)	0(0)	10(5)	2(1)	5(5)	3(5)	5(5)	0(0)	3(1)	4(4)	23(13)	2(1)	4(0)	11.4
75%	**S**	2(3)	0(0)	1(1)	1(1)	0(0)	4(1)	2(1)	1(1)	0(0)	1(1)	0(0)	2(0)	3(3)	4(2)	0(0)	3(0)	1.5
	FP	4(4)	0(0)	1(1)	3(1)	0(0)	6(1)	2(1)	1(1)	0(0)	1(1)	0(0)	2(0)	3(3)	6(4)	0(0)	3(0)	2
90%	**S**	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	1(0)	0(0)	0(0)	0(0)	0(0)	2(0)	2(0)	2(0)	0(0)	3(0)	0.6
	FP	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	1(0)	0(0)	0(0)	0(0)	0(0)	2(0)	2(0)	2(0)	0(0)	3(0)	0.6
100%	**S**	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	1(0)	0(0)	0(0)	0(0)	0(0)	2(0)	2(0)	2(0)	0(0)	3(0)	0.6
	FP	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	1(0)	0(0)	0(0)	0(0)	0(0)	2(0)	2(0)	2(0)	0(0)	3(0)	0.6

Table 2: Number of Intensions with Single and Fixed Point Inference at Different Confidence Levels

sion templates (calledBy, callersOf, and accessorsOf), is fairly even. We expected that intensions based on field accesses and method calls would constitute the most significant part of the inferred intensions, but because different concerns can describe varied subsets of the code of a system, we believe that a diverse set of intension templates can nevertheless be useful.

4.4 Threats to Validity

The main threat to the validity of our results is associated with the sample of concern mappings. We used concerns from a previous study, wherein the goal was to compare how different developers would describe the implementation of a high-level concern. The 16 mappings used may not be representative of concern mappings in general for two main reasons. First, they may be unique to the systems or features selected. Second, they were produced by people and as such are subject to the usual human factors (e.g., ability, experience, skill, and motivation of the subjects). However, given our intent to model code identified by developers, we believe that our use of human-generated concern models is more appropriate. The use of a relatively high number of sample concern mappings (given the cost of obtaining such samples) allows us to decrease the importance of any specific characteristic of a given sample, and thus to achieve a reasonable level of external validity. Finally, the mappings we use are publicly available, so their characteristics can be independently evaluated.

5. IMPLEMENTATION TRACKING

Using historical development data, we simulated how ISIS4J would have performed in a given situation to validate the usability of ISIS4J, and to gather insights into the kinds of evolutionary changes that could and could not be tracked with our approach. We studied 34 major releases of the jEdit editor, from version 4.0-pre1 (59 kLOC) to version 4.3-pre3 (92 kLOC). We selected the SYNTAX HIGHLIGHTING feature as the main concern of interest because it existed throughout all the studied releases and underwent several changes. This case study enabled us to answer the following research questions:

1. What changes to the implementation of the feature were tracked because of the inferred intensions?
2. What changes could not be tracked?
3. Does the intensional definition of the concern mapping enable us to better track the concern than its extensional definition?

5.1 Experimental Design

We manually created a concern mapping of the SYNTAX HIGHLIGHTING feature in the first and last releases of jEdit we considered. We explored the source code of jEdit using Eclipse, identifying fields and methods that we thought were the most relevant if this feature had to be modified. Because concerns typically do not exhibit crisp and explicit boundaries, we limited ourselves to no more than the 30 most relevant elements, to avoid systematically creating very large concern mappings. Thus, in the first and last versions, we identified respectively 24 and 27 elements. We used the mapping for the first version to infer the initial set of intensions. The mapping for the last version, not extended by ISIS4J or any other tools, enabled a comparison with the results produced by using ISIS4J to track the concern mapping's evolution.

To perform our case study, we automated the execution of ISIS4J. We selected a confidence threshold of 60% and inferred intensions until we reached a fixed point in the first version. We then used the same set of intensions throughout all other jEdit versions and did not try to infer new intensions in each version (since the intermediate concerns were not created by developers). Furthermore, we configured the experimental framework as follows:

Intension Conservation. As the system evolves from one version to another, program elements such as methods and fields can be added or removed. It is thus possible that an intension's confidence decreases to a point where it no longer meets the required threshold. In this case, we automatically flag the intension as disabled in the concern definition, but we keep the elements that were previously matched by that intension. Disabled intensions are automatically re-enabled if they reach the threshold again. For example, there might exist an intension matching all the five methods accessing a field f_1 in program version 1, and all those five methods are part of the concern. If there are ten methods accessing field f_1 in version 2, confidence drops to 50% (five methods are in the concern mapping out of ten existing methods); thus, the intension will no longer be enabled in the inferred set of intensions, but the original five methods are kept in the concern mapping. It should be noted that in our study, disabling an intension is not the same as simply removing it, since we do not attempt to re-infer intensions in intermediate versions.

Inconsistencies. It is possible that an element (e.g., a method) identified in the concern mapping got deleted in a later version. This situation introduces an *inconsistency* between the concern mapping and the version of the system because the mapping describes elements that do not exist. Our automated experimental framework removed inconsis-

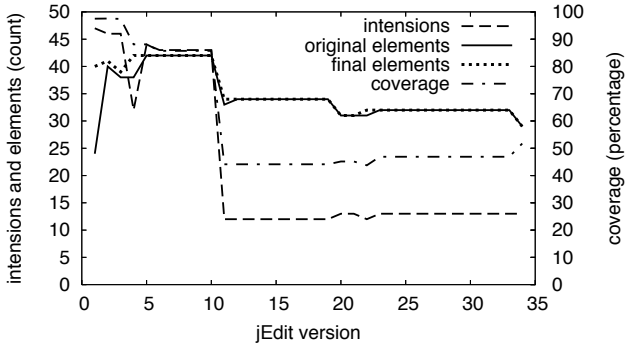

Figure 4: Intensions, Concern Elements and Coverage in Each Version of jEdit

tent elements from the concern mapping before applying the intensions on a version of the system. Elements that were renamed or moved are expected to be automatically discovered with our system provided that *1*) they are still matched by one of the intensions and *2*) the corresponding intension's confidence still exceeds the required threshold.

To summarize our experimental procedure, we manually produced a concern mapping for the first version of the system, and inferred a set of intensions for this mapping. Then, ISIS4J completed the mapping based on the inferred intensions. This process was repeated until a fixed point was reached. We then imported the modified concern mapping along with the final set of intensions in the next version of the system. Our framework then removed inconsistent elements from the mapping and disabled the intensions that no longer reached the threshold. ISIS4J then completed the mapping based on the intensions that were still enabled. We repeated this process until we reached the last version, and compared the final tracked concern mapping with the final manual concern mapping.

5.2 Results

Figure 4 shows the progress of the following measurements for the SYNTAX HIGHLIGHTING concern for each of the 34 considered versions of jEdit: the number of enabled intensions, the average number of concern elements per intension, the concern size (cardinality), as well as the ratio of elements in the concern that are matched by the enabled intensions (also called *coverage*). We started with 47 intensions in jEdit 4.0-pre1 and ended with 13 intensions in jEdit 4.3-pre3. We have a manual concern mapping sizes of 24 and 27 elements respectively, and an average of 3.66 and 2.62 concern elements per intension's extension respectively. Although the intensions in the first jEdit version cover 97% of the concern, this coverage drops to 52% after 34 versions.

Initial Intension-Based Completion. After ISIS4J inferred the intensions underlying our manual concern mapping, missing elements matching those intensions were added to the concern: the concern mapping size thus went from 24 elements to 40. All the elements added to the concern mapping were considered by the authors to be relevant to the SYNTAX HIGHLIGHTING feature as they contributed to this functionality. As we had limited the size of the manual concern mapping to a maximum of 30 elements, it was expected that ISIS4J would find other relevant elements. For example, the `Token` class, the main parsing unit of syntax highlighting, and two utility classes, `class GUIUtilities` and `TextUtilities`, were added to the concern description along with a subset of their methods. These elements had been omitted in the manual mapping in favor of elements using them.

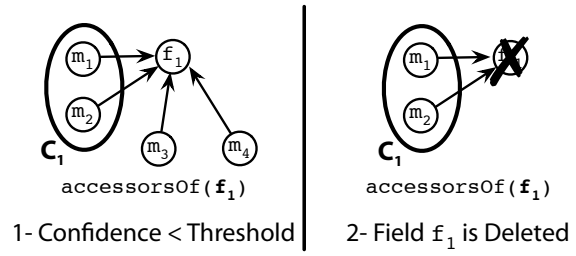

Figure 5: Disabling Intensions

Changes in Intensions Over Time. The overall evolution in the number of enabled intensions can be seen in Figure 4 (in the following discussion, release numbers are followed by the corresponding ordinal index of Figure 4). Although the number of intensions stays relatively stable up to jEdit 4.0-final (10), it drops down to 12 intensions in the following version (4.1-pre1, 11). From then on, the number of intensions stays mostly constant.

In general, the number of enabled intensions did not vary significantly between two consecutive jEdit versions. There are two main reasons why an intension might be disabled as illustrated by Figure 5: *1*) too many new elements were refactored or introduced in the program between two versions, hence lowering the intension confidence below the selected threshold, or *2*) the intension criterion (e.g., field f_1 in accessorsOf(f_1)), was deleted or renamed/moved. In the evolution of jEdit, 85% of the intensions were disabled because of the first reason, and 15% were disabled for the second reason.

There was only one major occurrence of intension disabling in the evolution of jEdit, indicated by the drop in the intensions count in Figure 4 (between jEdit 4.0-final (10) and jEdit 4.1- pre1 (11)). Three factors caused this drop. First, the deprecation of one class, `Buffer.TokenList`, removed five elements and disabled three intensions. Second, the move of the method `stringToToken` from class `XModeHandler` to `Token` disabled 14 intensions. This method was referring to fields only accessed by few other methods. When method `stringToToken` was moved and thus removed from the concern mapping, the intensions' confidence dropped below the threshold. As a result, ISIS4J disabled those intensions, the move was not captured and subsequent methods that referred to these fields were not included in the concern. The third factor related to the major drop in the intensions number is related to the SYNTAX HIGHLIGHTING feature extension. Since a significant number of new elements accessing intensions' criterion were introduced, this caused the intensions' confidence to drop below the threshold, disabling them.

Another interesting irregularity can be observed: a dent in the number of intensions around version 4.0-pre4 (4). The number of intensions drops from 46 to 32 in version 4.0-pre4 (4), and comes back to 44 in version 4.0-pre5 (5). We found this drop was caused by the refactoring of a method `loadStyles` that accessed fields used as criterion in various intensions. Figure 6 depicts how intensions were temporarily disabled (refactoring of method `loadStyles` is represented by methods m_1 and m_5). In version 4.0-pre4 (4), method m_1 was renamed to method m_5 causing some intensions confidence to drop below the threshold which disabled them (e.g.,

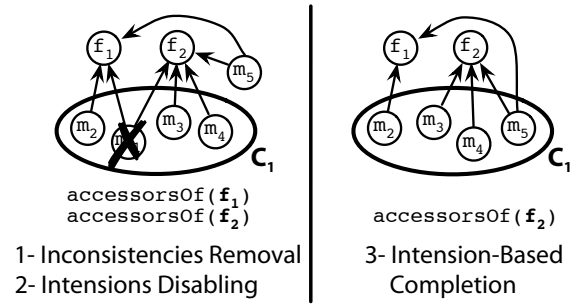

Figure 6: Intension Wrongly Disabled

accessorsOf(f_1)). Since other intensions stayed enabled (e.g., accessorsOf(f_2)), the new method m_5 was added during the intension-based completion. In the next version, 4.0-pre5 (5), the disabled intensions were re-enabled as their confidence reached the threshold again.

Comparison with the Final Manual Mapping. Our approach allowed us to detect four new elements of the SYNTAX HIGHLIGHTING concern throughout the evolution of jEdit. Examples of these elements include methods such as `init` in class `Chunk` or `setStyles` in class `TextAreaPainter`. Although none of those four elements was included in our final manual concern mapping, further inspection revealed that they were related to the SYNTAX HIGHLIGHTING feature and should have been included in the concern mapping in the first place. For instance, the `Chunk` class was introduced in version 4.1-pre1 (11) and replaced most of the `Token` class instantiations by extending it. The `setStyles` method, on the other hand, already existed in the first version of jEdit we studied. However, during its evolution, it became incohesive and redefined a token type, silently contributing to the SYNTAX HIGHLIGHTING feature. Arguably, those changes were not easy to manually track as they involved only a few lines of code in the existing program.

All elements in the final manual concern mapping that were not in the original manual mapping remained undiscovered by our approach. For example, a new hierarchy of classes implementing an interface `TokenHandler` was not included by ISIS4J. Manual investigation of the code revealed that most of the changes were introduced after version 4.1-pre1 (11) where the massive drop in the intension number occurred. Most of those changes could have been captured by the initial set of intensions provided they had not been disabled because of refactorings explained previously. Indeed, most new elements that were not captured actually accessed the disabled intensions' criterion.

Intensional vs. Extensional Concern Representation. We started with 24 concern elements in the first jEdit version. Considering that 17 elements became inconsistent during the evolution of jEdit and had to be removed from the concern mapping, we would have ended up with a concern mapping of only 7 elements if we had used an extensional concern mapping such as provided by ConcernMapper. Using the intensional concern definition that ISIS4J provided, we did better. First, ISIS4J did complete our initial manual concern mapping with 10 further elements, which indicates that completion of concerns using fixed point inference of intensions is helpful. Secondly, the 17 inconsistencies got removed while an additional 4 relevant elements were found. This gives us 18 concern elements in jEdit's final version rather than 7.

5.3 Discussion

The use of the jEdit historical development data allowed us to qualitatively evaluate the effectiveness of ISIS4J in tracking changes. The SYNTAX HIGHLIGHTING feature went through significant and different types of changes through its evolution: 17 elements were removed or refactored and the feature was extended in several ways. Our approach successfully captured feature extensions (e.g., the `Chunk` class), refactorings (e.g., the `loadStyles` method) and small but important changes (e.g., the `setStyles` method) that may have otherwise been difficult to track manually.

We also found that most of our inability to track changes (the small dent in the intension number in version 4.0-pre4 (4), the massive drop of intensions in version 4.1-pre1 (11) and the missed new elements between the manual mappings) were caused by the intensions' confidence level not reaching the 60% threshold. This is an indication that our approach could actually track changes even when major restructurings occur, provided that we lower the threshold. Future research will allow us to explore the tradeoffs associated with this strategy.

As we explained in Section 2.3, we introduced the threshold variable to ensure that we would only infer intensions that were representative of a concern's implicit structure even if it did not match it perfectly. ISIS4J also used the same threshold to enable or disable intensions between each version. We wanted to restrict the number of irrelevant elements introduced during the intension-based completion performed after the initial inference and between each version. However, as it turned out with the SYNTAX HIGHLIGHTING feature, the use of a threshold to enable/disable intensions inhibited the tracking of important changes, suggesting that it might be worth lowering it significantly or not using it at all once the initial intensions have been inferred. Indeed, in a manual review of the results, we found that 75% of the new classes present in the final manual concern mappings and not in the original one could have been found through the disabled intensions.

Finally, we have so far only investigated seven different intension templates. However, one can think of several other potentially promising templates. For example, an intension based on textual similarity would have found new concern elements as they came up during evolution of jEdit. More specifically, if we had an intension that would check for textual similarity, e.g., the same substring in method and field names, we would have identified all but 7 new elements as most of them had at least one of the following substrings in common: keyword, token, or rule.

5.4 Threats to Validity

Although our case study provides a valuable illustration of the potential benefit of the approach, its external validity is limited in that the case studied cannot be considered representative of the performance of the approach in general. In practice, two main factors will impact the usefulness of the approach: the characteristics of the concern analyzed, and the characteristics of the evolution of the code. In order to benefit from our system, a concern mapping must exhibit latent intensions that will be detected and potentially completed by ISIS4J. As seen in Section 4, not all concerns have such characteristics. Regarding the impact of the code, drastic changes to the structure of a system will inevitably limit the usefulness of ISIS4J. Our approach will be particularly

affected by changes involving the removal of elements used as intension criterion.

Investigator bias is also a threat to the internal validity of our case study as we selected the high-level concern and produced the original and final concern mappings. Our knowledge of ISIS4J could have influenced the results of our study. To reduce this threat, we only inspected the first and last versions of jEdit to ensure that we would not know about intermediate changes and involuntarily privilege parts of the code that could be tracked by our system.

6. RELATED WORK

Many approaches have been proposed to help locate and document concerns in source code, two activities that are intimately tied to our goal of tracking concerns in evolving source code.

Concern Code Location. A number of automated and semi-automated approaches have been proposed to help developers map high-level concerns to code entities. Concern location techniques complement our approach because they can be used to provide the initial concern mappings to ISIS4J. Concern location approaches often combine multiple analysis techniques such as information retrieval and branched call graphs [24] or latent semantic indexing and probabilistic ranking [16]. Other approaches have used dynamic analysis to find code entities that were used by certain features (e.g., [5, 7, 12]). For example, STRADA [4] is an Eclipse-based tool that captures and analyzes *scenario-based* execution traces in order to recover traceability links between development artifacts and features or requirements [5].

Static and dynamic *aspect-mining* techniques aim at locating crosscutting concerns in a project and could thus also provide the initial concern mappings to ISIS4J. The main difference between feature/concern location and aspect mining techniques is that while the former focus on locating the code associated with a specific feature, the latter identify code locations exhibiting a redundant, scattered nature. Various types of analysis have been proposed for aspect mining, including dynamic analysis [23], static analysis of call graph fan-in [14], and source code repository analysis [1].

Concern Documentation. Numerous approaches address the problem of documenting and representing concerns intensionally. With FEAT [18], a user can manually create intensional concern representations by selecting relationships among elements through the graphical user interface, instead of writing queries.

In contrast, the JQuery [10] exploration tool allows a user to select Java elements through queries. With JQuery, concerns can be described using different structural characteristics (e.g., hierarchy relationships, method calls, etc.) and regular expression matches. A different usage of queries was proposed by Mens and Kellens [15] with IntensiVE, the Intensional View Environment in which users can define views as a set of structurally similar classes and methods in a Smalltalk program. This is done according to logical rules that resemble ISIS4J intensions (e.g., all classes that declare a method 'accept' with a single parameter).

Aspect-Oriented Programming (AOP) [11] can also document *crosscutting* concerns intensionally: pointcut languages define naming and runtime patterns that ultimately describe the code elements where aspects will be injected.

Finally, PUMA [22], the generic query engine of the Concern Manipulation Environment (CME), included a proposal to enable users to define a concern intension using any kind of query languages that could potentially select any kind of artifacts (e.g., code elements, documents, etc.).

All of these approaches allow users to document concerns in a way that can be robust in the face of evolving software by relying on intensions. Although our technique also uses intensions to track concerns in evolving source code, the main innovation of our work is to provide a system allowing the automatic generation of intensions from purely extensional descriptions, such as the ones that could be produced by a feature location technique. By minimizing the cost of producing robust concern descriptions, we hope to make their use more prevalent in the maintenance of long-lived systems.

Implicit Structure Inference. Various techniques and tools have been developed to extract different types of implicit structure from development artifacts.

Marin et al. developed an aspect mining framework that identifies crosscutting concern (CCC) sorts, i.e., rule sets describing certain concern types [13]. For example, if all methods accessing a database also open and close a connection, the framework will detect a "Consistent Behavior" CCC sort. This sort can then be used to find all methods exhibiting this behavior or to visualize at a higher level the crosscutting concerns present in the software project. This framework uses techniques such as concept analysis and fan-in analysis to perform the rules inference. As opposed to CCC sorts that must match specific rule sets, ISIS4J can come up with any rules combination and is not restricted to one kind of concern.

Finally Ernst et al. [8] proposed an approach to discover function invariants to support software evolution by using dynamic analysis. Their technique can discover pre- and post-conditions such as `variable a` should equal the size of `array b` when entering a function. As is the case for ISIS4J, their technique benefits from fixed point inference since detected invariants are used to detect new ones. Although both approaches aim to support software evolution, ISIS4J performs its inference on coarser-grained elements and focuses on concern mappings instead of function invariants.

7. CONCLUSIONS

Most software change tasks require knowledge about the implementation of different concerns related to the task. This knowledge is often acquired through reverse-engineering efforts that may include automated feature location techniques. Unfortunately, detailed knowledge about a concern's implementation can become invalid as a code base evolves.

We proposed to make descriptions of a concern's implementation more robust to evolutionary changes by automatically inferring the *implicit structure* latent in a concern description. By detecting *intensions* (structural characteristics shared by different elements in a concern's description) using a rule engine, we can automatically check whether the intensions continue to hold in future versions of the source code, and to adapt the concern mappings accordingly.

Our initial investigation of this technique involved seven different intension templates based on common structural relations such as method calls and field accesses. Application of our technique on 16 independently-produced concern mappings showed that true intensions were rarely present in concern mappings, but could generally be inferred by relaxing the confidence (completeness) with which intension

templates had to be matched. A simulation of our approach on historical data involving 34 versions of the jEdit open-source editor showed how we can track the implementation of a concern much more robustly than by pruning documented elements that were no longer found in the code.

Although the usefulness of this approach is inherently tied to the characteristics of the concerns being tracked and to the evolution of the system, its complete automation means that it can be used at a minimal cost to development organizations (e.g., by running automatically upon committing changes to the code repository). Our ISIS4J system can thus provide an inexpensive technique that can be integrated in development environments to help preserve valuable knowledge acquired through reverse engineering activities.

Acknowledgments

The authors thank Ekwa Duala-Ekoko, Christian Lindig, and Alan Mycroft for valuable comments on the paper. This project was supported by the Natural Sciences and Engineering Research Council of Canada, the Royal Academy of Engineering (UK), and a Gates Scholarship.

8. REFERENCES

[1] S. Breu and T. Zimmermann. Mining aspects from version history. In *21st International Conference on Automated Software Engineering*, pages 221–230, 2006.

[2] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the European Conference on Object-oriented Programming*, pages 77–101, 1995.

[3] A. H. Eden and R. Kazman. Architecture, design, implementation. In *Proceedings of the 25th International Conference on Software Engineering*, pages 149–159, 2003.

[4] A. Egyed, G. Binder, and P. Grunbacher. STRADA: A tool for scenario-based feature-to-code trace detection and analysis. In *Companion to the Proceedings of the 29th International Conference on Software Engineering*, pages 41–42, 2007.

[5] A. Egyed and P. Grunbacher. Supporting software understanding with automated traceability. *International Journal of Software Engineering and Knowledge Engineering*, 15(5):783–810, 2005.

[6] S. G. Eick, T. L. Graves, A. F. Karr, J. Marron, and A. Mockus. Does code decay? Assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, 2001.

[7] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, 29(3):210–224, 2003.

[8] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st International Conference on Software Engineering*, pages 213–224, 1999.

[9] W. G. Griswold, J. J. Yuan, and Y. 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.

[10] D. Janzen and K. D. Volder. Navigating and querying code without getting lost. In *2nd International Conference on Aspect-Oriented Software Development*, 2003.

[11] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. 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, 1997.

[12] R. Koschke and J. Quante. On dynamic feature location. In *Proceedings of the 20th International Conference on Automated Software Engineering*, pages 420–432, 2005.

[13] M. Marin, L. Moonen, and A. van Deursen. A common framework for aspect mining based on crosscutting concern sorts. In *13th IEEE Working Conference on Reverse Engineering*, 2006.

[14] M. Marin, A. van Deursen, and L. Moonen. Identifying aspects using fan-in analysis. In *Proceedings of the 11th Working Conference on Reverse Engineering*, pages 132—141, 2004.

[15] K. Mens and A. Kellens. IntensiVE, a toolsuite for documenting and checking structural source-code regularities. In *10th European Conference on Software Maintenance and Reengineering (CSMR 2006)*, 2006.

[16] D. Poshyvanyk, Y.-G. Guéhéneuc, A. Marcus, G. Antoniol, and V. Fajlich. Combining probabilistic ranking and latent semantic indexing for feature identification. In *Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 77–101, 2006.

[17] M. P. Robillard. Tracking concerns in evolving source code: An empirical study. In *22nd IEEE International Conference on Software Maintenance*, pages 479–482, 2006.

[18] M. P. Robillard and G. C. Murphy. Representing concerns in source code. *ACM Transactions on Software Engineering and Methodology*, 16(1):3, 2007.

[19] M. P. Robillard, D. Shepherd, E. Hill, K. Vijay-Shanker, and L. Pollock. An empirical study of the concept assignment problem. Technical Report SOCS-TR-2007.3, School of Computer Science, McGill University, 2007.

[20] M. P. Robillard and F. Weigand-Warr. ConcernMapper: simple view-based separation of scattered concerns. In *2005 OOPSLA Workshop on Eclipse technology eXchange*, pages 65–69, 2005.

[21] E. Soloway, R. Lampert, S. Letovsky, D. Littman, and J. Pinto. Designing documentation to compensate for delocalized plans. *Communications of the ACM*, 31(11):1259–1267, 1988.

[22] P. Tarr, W. Harrison, and H. Ossher. Pervasive query support in the concern manipulation environment. Technical report, IBM Research, 2005. Report RC23343.

[23] P. Tonella and M. Ceccato. Aspect mining through the formal concept analysis of execution traces. In *11th Working Conference on Reverse Engineering*, pages 112–121, 2004.

[24] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. 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.