Representing Concerns in Source Code

MARTIN P. ROBILLARD
McGill University
and
GAIL C. MURPHY
University of British Columbia

A software modification task often addresses several *concerns*. A concern is anything a stakeholder may want to consider as a conceptual unit, including features, nonfunctional requirements, and design idioms. In many cases, the source code implementing a concern is not encapsulated in a single programming language module, and is instead scattered and tangled throughout a system. Inadequate separation of concerns increases the difficulty of evolving software in a correct and cost-effective manner. To make it easier to modify concerns that are not well modularized, we propose an approach in which the implementation of concerns is documented in artifacts, called concern graphs. Concern graphs are abstract models that describe which parts of the source code are relevant to different concerns. We present a formal model for concern graphs and the tool support we developed to enable software developers to create and use concern graphs during software evolution tasks. We report on five empirical studies, providing evidence that concern graphs support views and operations that facilitate the task of modifying the code implementing scattered concerns, are cost-effective to create and use, and robust enough to be used with different versions of a software system.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Modules and interfaces*; D.2.6 [**Software Engineering**]: Programming Environments—*Integrated Environments*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Documentation*; *restructuring, reverse engineering, and reengineering*; D.3.2 [**Programming Languages**]: Language Classifications—*Java*

General Terms: Algorithms, Documentation, Experimentation, Human Factors, Languages

Additional Key Words and Phrases: Separation of concerns, concern modeling, software evolution, aspect-oriented software development, Java

The work described in this article is based on an idea described in the paper entitled "Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies," published in the *Proceedings of the 24th International Conference on Software Engineering* (May, 2002). This research was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), a graduate fellowship from the University of British Columbia, IBM, and a McGill University start-up package.
Authors' addresses: M. P. Robillard, School of Computer Science, McGill University, 3480 University Street, McConnell Engineering Building no. 318, Montreal, QC, Canada, H3A 2A7; email: martin@cs.mcgill.ca; G. C. Murphy, Department of Computer Science, University of British Columbia, 2366 Mail Mall, Vancouver, BC, Canada, V6T 1Z4; email: murphy@cs.ubc.ca.

ACM Reference Format:
Robillard, M. P. and Murphy, G. C. 2007. Representing concerns in source code. ACM Trans. Softw. Eng. Methodol. 16, 1, Article 3 (February 2007), 38 pages. DOI = 10.1145/1189748.1189751 http://10.1145/1189748.1189751.

1. INTRODUCTION

Useful software systems need to change [Belady and Lehman 1976]. In general, the effort spent repairing, adapting, and enhancing software systems to ensure their continued value is significant because for each change task, a developer has to understand the existing software, modify it, and then validate the changes [Boehm 1976].

A software modification task often addresses several *concerns*. The term concern is loosely defined to represent anything that stakeholders of a software project may want to consider as a conceptual unit. Typical concerns in a software project include features, nonfunctional requirements, design idioms, and implementation mechanisms (e.g., caching). To make it easier to evolve the code base, software engineers generally try to keep concerns separate as they build a system [Parnas 1972]. Unfortunately, in practice, it is not possible to separate all concerns in a system due to a variety of reasons, including inadequate initial design, the limitations that programming languages impose on the decomposition of software systems [Kiczales et al. 1997; Tarr et al. 1999], the emergence of unforeseen concerns as a system evolves, and the decay of code structures following repeated changes [Eick et al. 2001; Belady and Lehman 1976; van Gurb and Bosch 2001]. As a result, many concerns end up scattered and tangled throughout the code of a system [Tarr et al. 1999]. Scattered concerns make it difficult for developers to reason about which pieces of code interact to implement a concern, and about how different concerns interact with each other. An incomplete understanding of a concern prior to a program change can cause a developer to make incorrect or inefficient modifications [Letovsky and Soloway 1986; Robillard et al. 2004], or modifications that do not respect an existing design [Parnas 1994].

To make it easier to modify concerns that are not well modularized, we propose an approach in which the implementation of concerns is documented in artifacts, called *concern graphs* [Robillard and Murphy 2002]. Concern graphs are abstract models that describe which parts of the source code are relevant to different concerns. A concern graph abstracts the implementation details of a concern by recording the key structure implementing it. By recording structure, a concern graph explicitly documents the relationships between the different sections of code that play a role in the implementation of a concern. A concern graph is based on a program model that can be extracted automatically from the source code or similar forms. By staying close to the source in its abstraction, a concern graph can remain tightly linked with the code.

To support our approach, we have created a tool, called FEAT, that helps developers build concern graphs semiautomatically as they investigate a program. Once a concern graph is available, the tool allows a developer to view only

those parts of a software system that are relevant to a concern, to view them easily in the context of the corresponding source code, and to perform analyses on the concern description (e.g., to see how two concerns overlap in the code).

We claim that concern graphs and their supporting technology are:

—*functional*, supporting views and operations that facilitate the task of modifying the code implementing scattered concerns;

—*cost-effective*, as they can be created as part of normal program investigation activities; and

—*robust*, as they can tolerate changes to a code base, and can thus be reused with different versions of a system. This property further reduces the relative cost of the approach, since a single concern graph can be used to support multiple tasks.

The need to document scattered concerns so as to support software evolution was identified by Soloway et al. [Letovsky and Soloway 1985; Soloway et al. 1988]. Unfortunately, traditional documentation such as that proposed by Soloway suffers from two principal drawbacks: It is costly to produce and difficult to maintain consistently with the source code. Program understanding and reverse engineering [Chikofsky and Cross II 1990] approaches have also been developed to help a developer discover the code related to a program change task (e.g., cross-reference search engines [Chen et al. 1990; Goldberg 1984; Object Technology International 2001; O'Brien et al. 1987; Sanella 1983], or feature location techniques [Wilde et al. 1992; Wilde and Scully 1995]). These approaches can help a developer track down the code relevant to a concern, but have a limited capacity to help developers preserve, view, and manage the knowledge discovered. As such, concern graphs not only help developers find and view information about the implementation of concerns in source code, but also enable them to preserve the information, opening new possibilities for linking documentation to source code. Aspect-oriented programming (AOP) approaches [Kiczales et al. 1997] aim to modularize crosscutting structure explicitly in programs. Concern graphs complement AOP by providing an inexpensive way to group scattered code, without requiring changes to the code base.

We have implemented our approach for the Java programming language, and performed multiple empirical studies to validate whether our approach meets the aforementioned claims. In Section 2 we motivate this research by presenting a small practical example. In Section 3 we present the formal model for concern graphs which is the foundation for our approach. Section 4 is a description of the existing tool support. In Section 5, we present the validation of the approach. Section 6 briefly discusses open questions arising from the research. We conclude with a discussion of related work in Section 7 and a summary of the article in Section 8.

2. MOTIVATION

To illustrate the challenge of finding and understanding the implementation of scattered concerns and to introduce our approach, we describe what a developer must consider and remember in order to extend a feature in the JHotDraw

drawing program.[1] JHotDraw is implemented in Java and comprises about 16kLOC of source code (not including comments and blank lines). In release 5.3 of JHotDraw, two different binary file formats are supported to store drawings to disk. The task of our developer is to modify the "save" feature of JHotDraw to support an additional file format. In this case, our concern of interest, SAVE FEATURE, corresponds to the code implementing the saving of drawings to disk.

Conceptually, the SAVE FEATURE concern is trivial. However, its implementation is not: The code implementing SAVE FEATURE is scattered throughout at least 35 classes, and interacts as well as intersects with other concerns, as the following three examples demonstrate.

—To understand where the action to save a drawing is triggered, the developer must discover the implicit invocation mechanism responsible for creating and triggering actions based on menu selections. In JHotDraw this mechanism, which we call the COMMAND concern, is implemented as a variation of the Command design pattern [Gamma et al. 1995], and involves at least 15 methods scattered in six classes (not counting concrete commands that have nothing to do with SAVE FEATURE).

—To understand how different storage formats are managed, the developer must investigate the details of the storage management mechanism. This STORAGE MANAGEMENT concern involves many different interactions between at least 15 methods scattered in four classes (e.g., `StorageFormatManager` and `StorageFormat`). These interactions are necessary to carry out such tasks as registering different file formats with a file chooser widget, and obtaining the writer object representing the correct storage format.

—To understand how to actually write a drawing to a file, a developer will need to realize that there exists a `Storable` interface, and that all the objects in a drawing that can be written to a file must implement a `write` method, which writes to a special type of object called `StorableOutput`. In version 5.3 of JHotdraw, the interface `Storable` is implemented by 24 different classes. We refer to this mechanism as the WRITING concern. In total, it is implemented by about 39 methods scattered in 31 different classes.

The challenge of modifying SAVE FEATURE stems from having to discover, understand, and keep track of separate, but interacting, *subsets* of the code associated with different high-level concepts (i.e., concerns). In practice, the STORAGE MANAGEMENT, COMMAND, and WRITING concerns all interact in the source code. For the change task, the developer will likely need to understand the details of each concern, including their interactions. As an example of interaction between the three concerns, Figure 1 shows the code of a method called `promptSaveAs()`. This method is the callback executed by the save command described in COMMAND, implements USER INTERFACE code (a fourth concern, lines 4 and 7), uses the STORAGE MANAGEMENT mechanism (lines 5, and 9–15), and calls into code implementing the WRITING concern (lines 14–15).

[1]What we call the JHotDraw program is technically the default application based on the JHotDraw drawing framework, version 5.3. http://www.jhotdraw.org/

```
     1  public void promptSaveAs()
     2  {
     3     toolDone();
UI   4     JFileChooser saveDialog = createSaveFileChooser();
SM   5     getStorageFormatManager().registerFileFilters(saveDialog);
     6
UI   7     if (saveDialog.showSaveDialog(this) == JFileChooser.APPROVE_OPTION)
     8     {
SM   9        StorageFormat foundFormat = getStorageFormatManager().
SM   10                                       findStorageFormat(saveDialog.
SM   11                                                         getFileFilter());
SM   12          if( foundFormat != null )
SM   13           {
SM,W 14              saveDrawing( foundFormat, saveDialog.getSelectedFile().
SM,W 15                                                      getAbsolutePath());
          ...
```

Fig. 1. Method `promptSaveAs()` of class `DrawApplication`. The abbreviations to the left of line numbers indicate the concerns implemented on each line: USER INTERFACE (UI), STORAGE MANAGEMENT (SM), and WRITING (W).

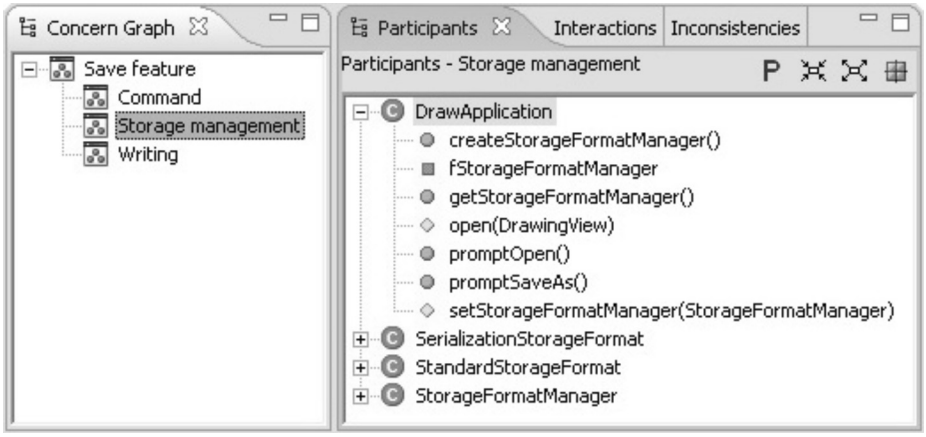

Fig. 2. Partial view of a concern graph for STORAGE MANAGEMENT in the FEAT tool.

To allow developers to capture subsets of the system relevant to concerns of interest, and to be able to work with the system in terms of the concerns, we developed a representation called concern graphs. A concern graph is built interactively while a developer investigates a program, shows only that subset of the program associated with the implementation of a concern, and allows developers to perform a number of analyses (e.g., to see how different concerns interact in the source code).

Figure 2 presents a partial view of a concern graph for SAVE FEATURE as displayed in FEAT, a tool we developed to support the concern graph approach. A concern graph represents the subset of a software system associated with a specific concern. Concern representations can be organized in a hierarchy (left window in Figure 2), where each subconcern specializes the parent concern. For example, the COMMAND concern represents that subset of the program

implementing specifically the save command in the Command design pattern.[2] Selecting any concern or subconcern on the left displays the subset of the program associated with the concern. For example, in Figure 2, the structure corresponding to STORAGE MANAGEMENT is displayed in the right window; only seven methods are visible from the 91 members and nine inner classes that make up the `DrawApplication` class. From this simplified view, it is also possible to perform other operations on concern graphs, such as to drill down to the source code and perform cross-reference queries and interaction analyses. For example, if the developer does not remember the role that method `open` plays in the implementation of the concern, selecting this method in FEAT will display (in a separate view) a link labeled "accessing `DrawApplication.fStorageFormatManager`." This link provides explicit information about the connection of `open` to the concern. Clicking on the link will automatically display and highlight the access to `fStorageManager`, which is buried in one of the 45 lines of the `open` method. Other aspects of FEAT that support software modifications are detailed in Section 4.

3. THE CONCERN GRAPH MODEL

In our approach, the program subsets relevant to a concern are called concern graphs. To provide a precise definition of a concern graph, we define it as a programming language-independent mathematical model based on relational algebra. This model underlies the operations that tools can provide on concern graphs. Except where indicated, the notation and definition of relational operators are taken from Schmidt and Ströhlein [1993].

3.1 Programs

The definition of a concern graph is linked to an underlying program model that specifies which information about a program can be captured by a concern graph. We model a program as a labeled, directed graph whose vertices are elements declared in the program and whose edges are the different types of relations between these elements.

Definition 1 (*Program Model*). A program model $P = (E, L, \{R_l\}_{l \in L})$ consists of a set E of program elements, a set L of relation labels, and a set of relations R_l, a distinct one for every label. Formally, $\forall\, l \in L\ \exists!\ R_l \subseteq E \times E$.

This definition ensures two important properties for the program model. First, the unique existential quantifier ensures that no two relations have an identical name. Second, although two vertices can have more than one edge between them, because the relations R_l are distinct, each edge must have a different label.

Definition 1 states that anything that can be known about a program according to our model must be expressed in terms of labeled relations between

[2]Note that the COMMAND box in Figure 2 only represents the code implementing the save command, not the entire Command design pattern. However, in a different context, it might be useful to describe the entire implementation of the Command design pattern in a concern graph.

elements. Given a concrete program P in some programming language, a model of this program is obtained by applying a language-specific *mapping function* M to the program. A model of program P, according to the mapping function M, can be represented by P_M when the specific mapping function matters (in the rest of this article, we will omit this subscript when the specific mapping function is irrelevant). Different mapping functions can be defined for a single programming language. In practice, the step of extracting a model from a program is performed via standard static analyses. Additional details concerning the specification of mapping functions can be found in a separate report [Robillard 2003c].

3.2 Fragments

Concern graphs are defined as a collection of building blocks, called *fragments*, which are assembled to form concern descriptions of increasing complexity. To facilitate the creation of concern graphs, we designed fragments to be conceptually similar to the queries that developers perform when investigating source code. Formally, a fragment describes a relationship between two sets of elements in a program model. A fragment consists of an *intension* and its corresponding *extension* (using terminology as applied to software engineering by Mens et al. [2002]). The intension of a fragment captures a high-level description of what we wish to describe about a program (e.g., "all the subclasses of class `C`"). The extension describes that actual subset of the program corresponding to the intension (e.g., "classes `A` and `B`"). The purpose of this redundancy is for fragments to tolerate software evolution, as will be shown in Section 3.5.

The definition of a fragment requires a *program subset specification* (or *specification*, for short). Specifications can be applied to a program model and result in a set of program elements.

Definition 2 (*Program Subset Specification Application*). Let $P = (E, L, \{R_l\}_{l \in L})$ be a program model and S be a program subset specification. The function apply(P, S) returns a set of elements $R \subseteq E$. We say that S is defined on P.

Technically, for a program model $P = (E, L, \{R_l\}_{l \in L})$, any specification resulting in a set $R \subseteq E$ constitutes a valid domain (or range). Although many different types of specification are possible, this article describes those two that have been the most useful to date:

—An extensionally-specified nonempty set of elements (e.g., $Dom = \{\texttt{A}\}$, $Ran = \{\texttt{A}, \texttt{C}, \texttt{D}\}$). Applying this type of specification produces the specified set.

—The universal domain (or range), represented by the symbol $\mathcal{E}$. The universal specification intuitively means "all applicable elements in a given program model." Applying this specification to a program model produces the set of elements for this model. In other words, for a program model $P = (E, L, \{R_l\}_{l \in L})$, apply$(P, \mathcal{E}) = E$.

To define the intension of a fragment, we use a specification for a *domain*, a relation label, and a specification for a *range*.[3] For example, to specify a fragment representing calls from method `a` to methods `b` and `c`, we would specify {`a`} as the domain specification, *Calls* as the relation label, and {`b`, `c`} as the range specification.

To describe the extension of a fragment, we define a projection operator on the specifications and label defining the intension of a fragment and a program model.

Definition 3 (*Projection Operator*). Let $P = (E, L, \{R_l\}_{l \in L})$ be a program model, Dom and Ran a domain and a range specification, and $l \in L$.

$$\text{proj}(Dom, l, Ran, P) = \text{apply}(P, Dom) \triangleleft R_l \triangleright \text{apply}(P, Ran).^4$$

In other words, the projection operator takes the intension of a fragment (a domain specification, relation label, and range specification) and a program model, and produces the relation corresponding to the intension for the specific program model. In practice, the projection operator corresponds to a query mechanism, and the result of the projection is the query result.

We now have all of the tools required to formally define a fragment.

Definition 4 (*Fragment*). Let $P = (E, L, \{R_l\}_{l \in L})$ be a program model. Let Dom and Ran be two specifications defined on P, and $l \in L$. We define a fragment as $f_P = (Dom, l, Ran, Proj)$, where $Proj = proj(Dom, l, Ran, P)$. We say that f_P is defined on P.

Creating a fragment thus consists in specifying a domain, a range, and a relation label, and applying the projection operator on a program model. The result describes a subset of the program model.

We illustrate different possibilities for fragment specification through a series of examples based on a mapping function J for the Java language. The J mapping function only considers types (classes and interfaces), and methods. The relationships modeled are restricted to the identity relation ($\mathcal{I}$), the declarative structure of the program (*Declares*), and static method calls (*Calls* and *CalledBy*).[5] Based on this mapping function, we can specify fragments on program P1 (Figure 3).

A model for program P1 is shown in Figure 4.[6]

To describe a single program element as a fragment, we use the identity relation $\mathcal{I}$. In the examples, fragments are named with a phrase describing their intension.

$$\textbf{Class A} := (\{\texttt{A}\}, \mathcal{I}, \{\texttt{A}\}, \{(\texttt{A}, \texttt{A})\})$$

[3]The domain and range of a fragment are specifications, not to be confused with the relational operators dom() and ran().

[4]The symbols $\triangleleft$ and $\triangleright$ denote the domain and range selection operators, respectively.

[5]This mapping function is simple for illustrative purposes. The model we support in practice comprises many additional element types and relations.

[6]The notation has been simplified by omitting the full qualification of Java names and the parentheses in method signatures.

```
public class A
{
   public static void b(){}
   public static void c(){ c(); b(); D.f(); }
}

class D
{
   public static void e() { f(); }
   public static void f() {}
}
```

Fig. 3. Program P1.

Model $\mathbf{P1_J}$
$E = \{\mathtt{A}, \mathtt{b}, \mathtt{c}, \mathtt{D}, \mathtt{e}, \mathtt{f}\}$
$L = \{\mathcal{I}, Declares, Calls, CalledBy\}$
$R_{\mathcal{I}} = \{(\mathtt{A}, \mathtt{A}), (\mathtt{b}, \mathtt{b}), (\mathtt{c}, \mathtt{c}), (\mathtt{D}, \mathtt{D}), (\mathtt{e}, \mathtt{e}), (\mathtt{f}, \mathtt{f})\}$
$R_{Declares} = \{(\mathtt{A}, \mathtt{b}), (\mathtt{A}, \mathtt{c}), (\mathtt{D}, \mathtt{e}), (\mathtt{D}, \mathtt{f})\}$
$R_{Calls} = \{(\mathtt{c}, \mathtt{b}), (\mathtt{c}, \mathtt{c}), (\mathtt{c}, \mathtt{f}), (\mathtt{e}, \mathtt{f})\}$
$R_{CalledBy} = \{(\mathtt{b}, \mathtt{c}), (\mathtt{c}, \mathtt{c}), (\mathtt{f}, \mathtt{c}), (\mathtt{f}, \mathtt{e})\}$

Fig. 4. Model $\mathrm{P1_J}$.

We can also describe a single method call as a fragment:

$$\textbf{c calls b} := (\{\mathtt{c}\}, Calls, \{\mathtt{b}\}, \{(\mathtt{c}, \mathtt{b})\})$$

We can describe slightly more elaborate interactions using the universal range. For example, to capture all members of class `A`, we specify:

$$\textbf{Members of A} := (\{\mathtt{A}\}, Declares, \mathcal{E}, \{(\mathtt{A}, \mathtt{b}), (\mathtt{A}, \mathtt{c})\})$$

We can also use the universal range to capture all the callers of `f`.

$$\textbf{Callers of f} := (\{\mathtt{f}\}, CalledBy, \mathcal{E}, \{(\mathtt{f}, \mathtt{c}), (\mathtt{f}, \mathtt{e})\})$$

The last operation we define on fragments is the *participants* operation. For any fragment, this produces a set of elements involved in the fragment.

Definition 5 (*Fragment Participants*). Let $f = (Dom, l, Ran, Proj)$ be a fragment.

$$\text{participants}(f) = \text{dom}(Proj) \cup \text{ran}(Proj)$$

3.3 Concerns

With fragments, it is possible to express different interactions between program elements. To attach a meaning to a collection of fragments, we define the notion of *concern representation* (or simply, concern) recursively as a named set of fragments and a set of *subconcerns*. The recursive definition is intended to allow developers to break complex concerns into simpler elements, following the divide-and-conquer principle.

Definition 6 (*Concern*). A concern (n, F, C) defined on P is a tuple comprising an identifying name n, a set of fragments $F = \{f_1, f_2, \ldots\}$ defined on P, and a set of concerns defined on P, $C = \{c_1, c_2, \ldots\}$. The definition of a concern must be acyclic.

The only constraint on the composition of fragments into a concern representation is that all the fragments be defined on the same program model P. We then say that a concern is defined on P. The only constraint on the inclusion of concerns as subconcerns of a parent is that the definition of a concern be acyclic. Either or both F and C can be the empty set. A fragment in F can also be in any subconcern c. Besides the constraints stated previously, fragments and concerns are composed freely into other concerns based on the needs of a user of the representation. There is no restriction on the names used to distinguish concerns, since this element does not impact the reasoning that can be done with concerns.[7] A root concern, not included in any parent concern, represents the broadest abstraction for a particular concern. It is called a concern graph.

The participants of a concern are defined as any element participating in a fragment within the concern.

Definition 7 (*Concern Participants*). Let $c = (n, F, C)$ be a concern, where $F = \{f_1, f_2, \ldots, f_p\}$ is a set of fragments and $C = \{c_1, c_2, \ldots, c_q\}$ a set of concerns. The participants of c are defined as:

$$\text{participants}(c) = \bigcup_{i=1}^{p} \text{participants}(f_i) \cup \bigcup_{j=1}^{q} \text{participants}(c_j)$$

3.4 Concern Interaction Analysis

Two concerns can potentially overlap or be related. Given two concerns defined on a common program model, we define their common participants as any program element participating in both concerns.

Definition 8 (*Common Participants*). Let c_1 and c_2 be two concerns defined on the same program model. The set of common participants is defined as:

$$\text{common}(c_1, c_2) = \text{participants}(c_1) \cap \text{participants}(c_2)$$

Even if two concerns have no element in common, they can still interact. We define the interaction between two concerns defined on a common program model as the set of all modeled relations between an element in one concern and an element in the other.

Definition 9 (*Concern Interaction*). Let c_1 and c_2 be two concerns defined on a program model $P = (E, L, \{R_l\}_{l \in L})$. The interaction between c_1 and c_2 is defined as:

$$\begin{aligned}\text{interaction}(c_1, c_2) = \{(\{x\}, l \in L, \{y\}, \{(x, y)\}) \mid\ & x \in \text{participants}(c_1) \wedge \\ & y \in \text{participants}(c_2) \wedge \\ & (x, y) \in R_l\}\end{aligned}$$

[7]For practical reasons, additional constraints can be imposed by different implementations of the model (e.g., a requirement for unique names).

In other words, the interaction between two concerns is a set of fragments representing the relations between the participants of one concern and the participants of the other. It is important to note that this definition is directional (noncommutative). The motivation behind this decision is to document concern interactions as a set of fragments (which are directional). There are many practical advantages to this definition, including the treatment of concern interactions as concerns themselves in supporting tools. Moreover, the directionality of the concern interaction operation is mitigated in practice by the fact that typical mapping functions will include the transpose of almost all relations.

The interactions between participants can also be defined for a single concern, enabling us to establish a closure of interactions between the participants of a concern. Specifically, given a concern c, the operation interaction(c, c) produces a set of fragments representing all the interactions between participants of c.

3.5 Inconsistency Management

Because concern graphs are defined on a specific program model, any change to the program impacting the model may render a concern graph inconsistent with the new program model corresponding to the changed source code. Such inconsistencies can be formally defined through a Boolean function IsInconsistent (x, P), where P is a program model and x is either a fragment or a concern.

Definition 10 (*Fragment Inconsistency*). Let $P_1 = (E_1, L, \{R_{1,l}\}_{l \in L})$ and $P_2 = (E_2, L, \{R_{2,l}\}_{l \in L})$ be the models corresponding to two versions of a program produced with the same mapping function. Let $f_{P_1} = (Dom_1, l, Ran_1, Proj_1)$ be a fragment defined on P_1.

$$\begin{aligned}\text{IsInconsistent}(f_{P_1}, P_2) \iff\ & \text{apply}(P_2, Dom_1) \not\subseteq E_2 \vee \\ & \text{apply}(P_2, Ran_1) \not\subseteq E_2 \vee \\ & Proj_1 \neq \text{proj}(Dom_1, l, Ran_1, P_2).\end{aligned}$$

In other words, a fragment is inconsistent with a program model if either its domain or range is inconsistent, or if its projection does not match the equivalent projection on the new program model. This support for detection of inconsistencies is the main justification for the existence of projections. Fragment projections store only the minimal subset of a program model required to check for inconsistencies with a different model.

Given the preceding definitions, we can define the inconsistency operator for concerns.

Definition 11 (*Concern Inconsistency*). Let P_1 and P_2 be the models corresponding to two versions of a program produced with the same mapping function. Let $c = (n, F, C)$ be a concern defined on P_1.

$$\text{IsInconsistent}(c, P_2) \iff \bigvee_{f \in F} \text{IsInconsistent}(f, P_2) \vee \bigvee_{c' \in C} \text{IsInconsistent}(c', P_2)$$

Finally, it is possible to define, at the level of the concern graph model, the conditions in which an inconsistency between a fragment and a model can be

automatically repaired, as well as the semantics of the repair operation. This way, we can ensure a common behavior for inconsistency repair across programming languages and tools supporting the concern graph model. A repairable fragment is defined as one for which both the domain and range are consistent (i.e., the fragment is only inconsistent in terms of its projection on the new program model). The repair operation is modeled as a function, taking as parameters a repairable program fragment defined on a model and inconsistent with a second model, and the second model. The operation returns a fragment with the same intension as the original, but which is consistent with the second program model.

Definition 12 (*Fragment Repair Operator*). Let $P_1 = (E_1, L, \{R_{1,l}\}_{l \in L})$ and $P_2 = (E_2, L, \{R_{2,l}\}_{l \in L})$ be the models corresponding to two versions of a program produced with the same mapping function. Let $f_{P_1} = (Dom_1, l, Ran_1, Proj_1)$ be a fragment defined on P_1 such that:

$$\begin{aligned} &\text{IsInconsistent}(f_{P_1}, P_2) \wedge \\ &\text{apply}(P_2, Dom_1) \subseteq E_2 \wedge \\ &\text{apply}(P_2, Ran_1) \subseteq E_2 \end{aligned}$$

We have

$$\text{repair}(f_{P_1}, P_2) = (Dom_1, l, Ran_1, \text{proj}(Dom_1, l, Ran_1, P_2)).$$

In informal terms, the repair function simply replaces the inconsistent projection of a fragment with a new projection that is consistent with the second program model. The practical implications of the inconsistency management support intrinsic to concern graphs are described in detail in Section 4.3.1.

4. TOOL SUPPORT FOR CONCERN GRAPHS

To support the concern graph approach, we developed a tool called the feature exploration and analysis tool (FEAT) [Robillard 2003a]. FEAT allows developers to create and manage concern graphs, and to view and modify the code of a system through concern graphs. To integrate these tasks with normal software development activities, we built FEAT as a plug-in for the Eclipse platform [Object Technology International 2001]. Eclipse is an integrated development environment with an architecture that supports the addition of modules, called plug-ins, that add to the environment's functionality. The standard distribution of Eclipse includes a set of plug-ins that provide extensive support for development in the Java programming language.

4.1 Anatomy of a Concern Graph in FEAT

In Eclipse, collections of views associated with a specific activity (e.g., debugging) are called *perspectives*. We created a new perspective, called the FEAT perspective, which shows a concern graph in decreasing levels of abstraction. Figure 5 shows the general layout of the FEAT perspective.

The Concern Graph View (area 1) shows the name of each concern in the concern graph hierarchy (see Section 3.3). From this view, users can create new

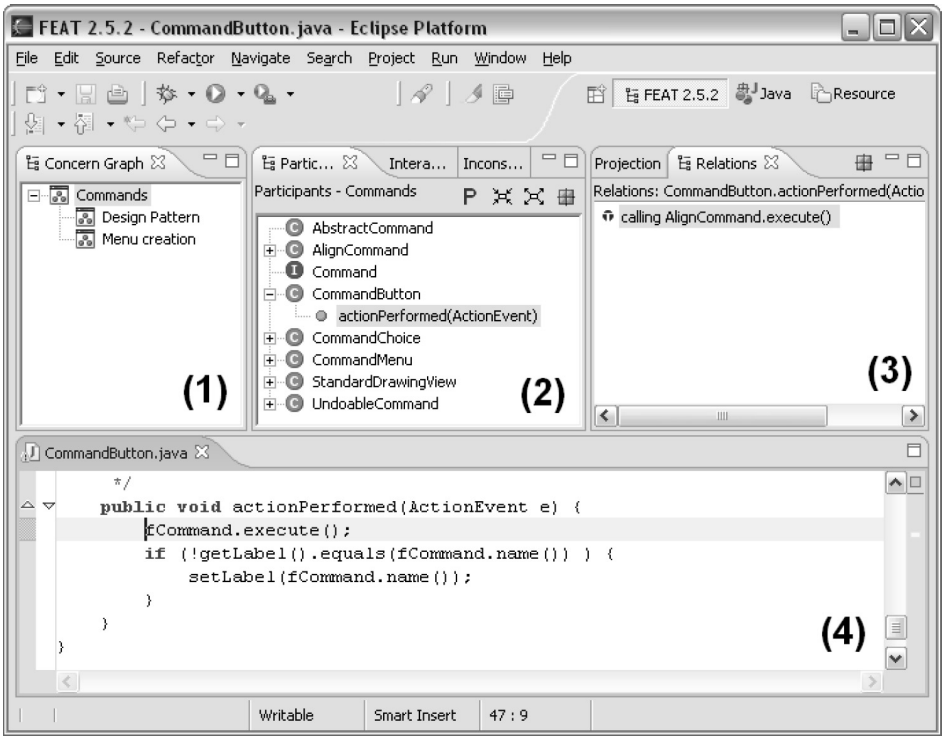

Fig. 5. The FEAT Perspective.

subconcerns, delete existing concerns, and move concerns in the hierarchy.[8] Selecting any concern name in the Concern Graph View displays all of the participants for the corresponding concern (see Definition 7, Section 3.2). We call the concern currently selected in the Concern Graph View the *active* concern. FEAT displays the participants for the active concern in the Participants View (area 2).

In the Participants View, the participants for a concern are displayed as a set of trees with participant classes at the root of the trees; participant members are displayed as children of their declaring class. Double-clicking on any participant shows its declaration in a Java editor (area 4). Selecting a participant shows all of the relations between this participant and any other participant in the active concern in the Relations View (area 3). For example, Figure 6 shows the relations for participant `Command.execute()`. The icon to the left of a relation indicates whether a relation is part of a fragment explicitly added by a user (dot), or was identified through the intraconcern analysis described in Section 3.4 (question mark).[9] Relations identified through intraconcern analysis are displayed, but are not part of a concern. A user can add these relations to a concern. Clicking on

[8]In our current implementation of the concern graph model, concerns can only have one parent. Although concerns constitute a graph of interactions between program elements, the hierarchical organization of concerns is a tree.

[9]Interconcern analysis is performed using a different view, described in Section 4.3.

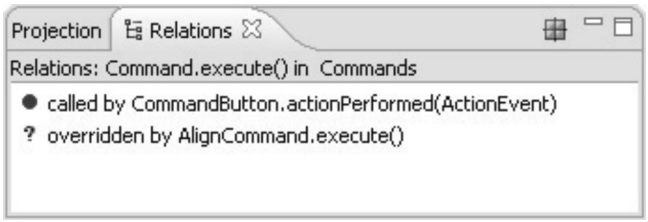

Fig. 6. The Relations View.

any relation shows the source code corresponding to the relation. For example, clicking on the relation `called by CommandButton.actionPerformed(ActionEvent)` will bring up the code of `actionPerformed(ActionEvent)` in the editor area and highlight the call to `Command.execute()`.

4.2 Building a Concern Graph

To build a concern graph, a developer first uses an Eclipse wizard to create an empty concern graph. This action associates the concern graph with a Java project, and builds a program database representing the program model that the concern graph will rely on (see Section 3.1). Once an empty concern graph is created and a program model available, the developer can query the program model to build a concern graph.

4.2.1 *Creating the Program Model.* The mapping function used by FEAT to generate a program model considers all the *types*, *methods*, and *fields* in a project as the elements in E, and a set of 23 relations between these elements (described in the FEAT user manual [Robillard 2003a]).

Local (intramethod) elements, such as method parameters and local variables, are not included in the model. We decided not to include these elements because we wanted to establish a practical bound on the size and complexity of the models required to define concern graphs, and because we are mostly interested in capturing concerns presenting interactions not limited to a module. Specifically, since elements such as local variables cannot be referenced by elements outside the method, they are not useful for describing a concern scattered in multiple methods. As the designers of the C information abstractor tool have noted, "[d]etails of interactions between local objects are ignored because they are only interesting in a small context" [Chen et al. 1990, p. 326]. We followed a similar philosophy in elaborating the design of our mapping function: In the absence of empirical data supporting the usefulness of modeling intramethod elements, we left these elements out in order to provide a more efficient and scalable approach. The performance of our model extraction mechanism is discussed in Section 4.4.

4.2.2 *Querying the Concern Model.* To minimize the cost of building concern graphs, FEAT enables developers to build concern graphs as a by-product of program investigation. FEAT supports queries that directly correspond to a fragment which has a universal range (see Section 3.2). For example, a query

Fig. 7. Query results in the Projection View.

to determine all of the callers of a method `m()` is modeled as the fragment intent

`m()` `CalledBy` `ALL`.[10]

A user performs a FEAT query by right-clicking on a Java element in any view in Eclipse showing Java elements (including FEAT Views), and by choosing a relation from a menu. Internally, FEAT queries are built and managed as fragments: Performing a query consists of applying the projection operator defined in Section 3.2 on the database stored by the plug-in. The results of a query correspond to the projection of the fragment that represents the query. Query results are displayed in the Projection View. The Projection View is the main view used to investigate the code in FEAT. It is shown in the same area (area 3 in Figure 5) as the Relations View in the FEAT Perspective: Selecting a tab at the top of the area allows a user to switch between views. Figure 7 shows the results of a query in the Projection View. Query results are displayed in a tree representing the projection of a fragment. The elements in the tree above the relation node represent the domain of the projection, while those below represent the range of the projection. In our example, the `execute()` method is called by methods in five different classes, including `CommandButton`. Double-clicking on any element in the Projection View displays its declaration in an editor. Selecting an element displays the source code for only the relation. For example, selecting method `actionPerformed(ActionEvent)` will display the line in the `actionPerformed(ActionEvent)` method where `execute()` is called. From within the Projection View, it is also possible to add elements and relations to a concern graph. To add a single element to the active concern, a user can right-click on any element in the view and select `Add element to concern`. This action will result in the addition of a single element to the active concern; the element is expressed as a fragment using the identity relation. To add a query result and the corresponding relation to the active concern, a user can select any range

[10]In this section, we use the keyword "ALL" to represent the universal range specification $\mathcal{E}$.

element (i.e., below the relation node in the tree), right-click and select `Add relation to concern`. This action will add to the active concern a fragment consisting of the element queried (as the domain), the relation queried, the element selected in the Projection View (as the range), and again, the element selected (as the projection). For example, in the case of Figure 7, if a user right-clicks on `actionPerformed(ActionEvent)` and selects `Add relation to concern`, the fragment intent

`Command.execute()` `Called By` `CommandButton.actionPerformed(ActionEvent)`

will be added to the concern. Such fragments are very common in practice. They describe a single structural relation between two elements, and are composed of a singleton domain and a singleton range. We call such fragments *primitive*. It is also possible to add the entire query result to the active concern though a menu in the tool bar of the Projection View. In this case, the fragment that is added to the active concern consists of the element queried (as the domain), the relation queried, the universal range, and the projection corresponding to the query results. Although the formal concern graph model supports use of the universal specification for the domain of a fragment as well as its range, our current version of FEAT only supports specifying a universal range. We made this decision so as to simplify the FEAT user interface, based on experimentation with early prototypes. For manual concern building such as currently done with FEAT, the specification of a fragment with a universal domain can be emulated with the transpose of a relation. Use of concern graphs in different contexts (e.g., automatic generation) may require the flexibility offered by the general formal model.

4.3 Using Concern Graphs

The main use of concern graphs is to give developers a single, focused view of the subset of a program implementing a concern. This view is provided by the Participant View. In addition, information about the relations between different concern participants is made explicit in the Relations View. All FEAT views, except the Concern Graph View, support a direct connection to the source code, allowing a developer to execute changes to the source directly in the context of a concern graph. Finally, FEAT allows developers to analyze how different concerns interact (see Section 3.4). The results of the interaction analysis appear in a view called the Interactions View, which overlaps the Participants View. The Interactions View shows the participants of two user-selected concerns side-by-side. Participants common to both concerns are annotated with a red diamond. Participants in one concern that are directly related to any participant in the other concern through a relation supported by the model are annotated with a yellow diamond. Figure 8 shows an example of the Interaction View. The view shows two concerns, `Command` and `GUI`, and highlights with diamonds the fact that two methods in the GUI concern call the constructor of class `ChangeAttributeCommand`, which is in the `Commands` concern. Clicking on any element with a yellow diamond will explicitly show the relation between the two concerns in the Relations View (area 3 of Figure 5).

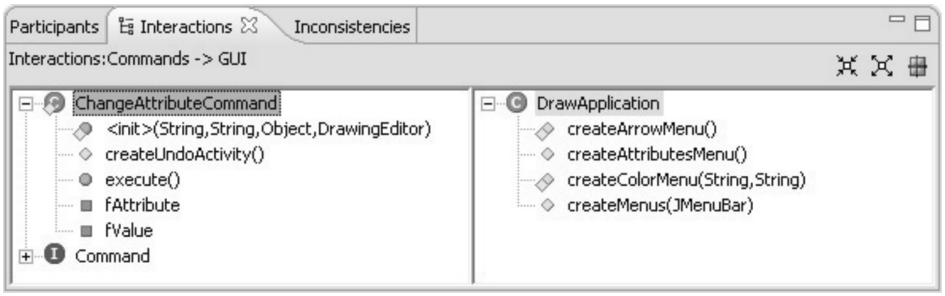

Fig. 8. The Interactions View.

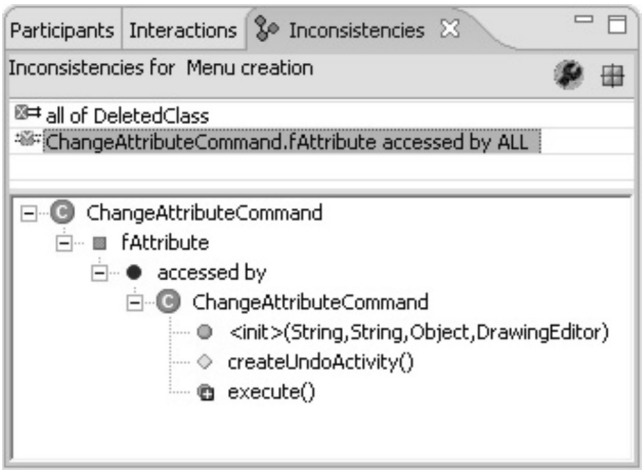

Fig. 9. The Inconsistency View.

4.3.1 *Reusing Concern Graphs.* Concern graphs are an abstraction of the source code of a system. When the source code associated with a concern graph is modified, the concern graph can become inconsistent with the system. To ensure that our approach is robust enough to remain usable in such situations, the FEAT tool is tolerant of inconsistencies between a concern graph and the source code. When a concern graph is loaded into FEAT, and any time the source code changes, FEAT performs an inconsistency check for each fragment. Checking for inconsistencies consists of applying the IsInconsistent function defined in Section 3.5 (Definition 10) to each fragment. Even in the case where inconsistencies are detected, FEAT functions as usual: Participants in consistent concerns are displayed and can be queried and analyzed. However, the participants and relations for any inconsistent fragment are not displayed in the Participants View and Relations View. To indicate that inconsistencies were detected, any concern containing one or more inconsistent fragments is annotated with an icon in the Concern Graph View. Additionally, it is possible to view, query, and repair inconsistent fragments in the Inconsistency View. Figure 9 shows the Inconsistency View listing two different inconsistencies. Inconsistencies are identified by the name of the inconsistent fragment. FEAT recognizes

three different types of inconsistencies based on its specific implementation of the model:

—*Primitive inconsistency*: The relation captured by a primitive fragment (see Section 4.2.2) does not exist in the source code. Specifically, a primitive inconsistency is detected when the IsInconsistent function applied to a primitive fragment returns true because any clause in Definition 10 is true. Primitive fragments represent individual selections from query results. These types of inconsistencies are not automatically repairable.

—*Domain inconsistency*: The domain of the fragment is inconsistent. Specifically, a domain inconsistency is detected when the IsInconsistent function returns true because the domain element of a fragment with a universal range does not exist in the current model (see Definition 10, Section 3.5). These types of inconsistencies are not automatically repairable.

—*Projection mismatch inconsistency*: The projection of the fragment does not match the current source code. Specifically, a projection mismatch inconsistency is detected when the IsInconsistent function returns true because the third clause in Definition 10 is true, but the two other clauses are false. These types of inconsistencies are automatically repairable (see Definition 12, Section 3.5).

Clicking on any fragment name in the inconsistency list displays the inconsistent fragment in a tree structure in the lower part of the view. Any element in the inconsistent fragment which exists in the source code can be displayed in an editor or queried, as in the Participants View or Projection View. This display allows users to investigate the relations between an element in an inconsistent fragment and the rest of the code base. As a result of such queries, a user may decide to add to the concern description based on information in the inconsistent fragment. Elements in the lower part of the Inconsistency View are annotated with an icon denoting whether the element does not exist, whether the corresponding relation exists in the code but not in the concern graph, or whether it exists in the concern graph but not in the code. For example, Figure 9 shows two inconsistent fragments. The first in the list indicates that class `DeletedClass` and all of its members were recorded in a concern graph, but do not currently exist in the project. The second inconsistent fragment in the list has the intension:

`ChangeAttributeCommand.fAttribute accessed by ALL`

Since this fragment is selected, its extension is shown in the detailed view, with method `execute` annotated with a "+" icon. This icon indicates that one of the current accesses to `fAttribute` (that from method `execute()`) is recorded in the concern graph, but does not exist in the current version of the project.

The Inconsistency View also allows a user to make an inconsistent concern graph consistent with the source code. First, any element in the Inconsistency View that exists in the current program model can be investigated using FEAT's query mechanism. A scenario illustrating how this technique can help evolve concern graphs is presented in Section 5.5. Second, right-clicking on any fragment in the list of inconsistent fragments will bring up a pop-up menu with the

item `Repair`. Repairing a repairable fragment will synchronize the fragment with the source code according to the algorithm of Section 3.5. For example, the fragment selected in Figure 9 is inconsistent due to a projection mismatch, and as such, can be automatically repaired. Repairing a nonrepairable fragment will remove the fragment from the concern graph.

4.4 Implementation

Besides the user interface described in the previous sections, the architecture of FEAT is comprised of two components: the model and the analyzer.

The model component supports the runtime representation of a concern graph, including loading and storing a concern graph, and tolerating inconsistencies between a concern graph and the source code through a mechanism of pollution markers inspired by the work of Balzer [1991]. It is important to note that the model component only represents a *concern graph* model, and does not contain any of the program information forming the *program* model. The latter is stored in the analyzer component.

The responsibility of the analyzer component is to produce a model for a program to support queries on this model, and to support mapping fragments to the corresponding source code. In our current version of FEAT, a user can only associate a single concern graph to a given program model. As a consequence, a user wishing to work with multiple concerns must do so in the context of a single concern graph. In practice, this simplifying design decision seems to have little impact on the usability of FEAT and we have not yet received any request to change this aspect of the tool.

The analyzer component is designed to optimize the speed of FEAT queries at the cost of an initial model extraction time. The FEAT analyzer produces a model of a program by executing a single pass through the abstract syntax tree (AST) of every Java file in the project associated with a concern graph. When scanning the AST of Java source files, the analyzer searches for instances of the relations supported by the mapping function. When a relation between two elements is identified, the analyzer stores both the relation and its transpose in an in-memory database. Our choice of a source code-based approach for the production of a program database stems from the convenience offered by Eclipse's incremental compiler, and is independent from any fundamental properties of concern graphs. For example, our initial FEAT prototype used bytecode analysis to produce the program database [Robillard and Murphy 2002]. It is important to note that the use of source code has a limited practical impact on the overall approach, since it is still possible to add any program element to a concern graph, whether it is defined in source code or in bytecode. Our program database also includes relations from source elements to bytecode elements. In the case of relations from bytecode elements (which are not supported), we presume that these are of limited applicability, since users of FEAT cannot change the corresponding code. There are a few cases (such as callbacks from framework methods) where the information could be useful, but so far, we have not found this to be a severe enough limitation to warrant redesign of the analyzer component.

Table I. Empirical Studies and Claims Addressed

Component/Claim	1. Functional	2. Cost-effective	3. Robust
1. AVID	⋆		
2. Jex		⋆	
3. Redback		⋆	
4. jEdit	⋆		
5. ArgoUML			⋆

The static analysis required to extract the model of a program, and the size of the database produced, both impose practical limits on the size of the programs analysable by FEAT. To allow FEAT to work on large programs, developers can control the scope of the analysis. When creating a new concern graph for a project, a developer can select from a list of all source packages for a project just those that should be included in the program model database. Elements declared in packages left out of the analysis can still be viewed and used in FEAT, but their source code is not analyzed. This flexibility allows users to remove basic libraries and other elements that they know are not involved in the concerns they are analyzing, reducing the storage and computation load on the tool.

Because the analyzer performs a single pass through the source code, the time required for model extraction increases linearly with the size of the source code analyzed, with an origin close to zero and a slope of just under 1.1s/kLOC.[11]

To mitigate the cost of model extraction, the FEAT tool updates the model incrementally after an initial extraction. In other words, after an initial extraction, every time a source file is changed, FEAT analyzes the changes and updates only the affected relations in the model database. This technique avoids costly periodic reextractions of the model.

5. VALIDATION

We make three claims about concern graphs. Our first claim is that concern graphs are *functional* in that they facilitate the task of modifying the code implementing scattered concerns. Our second claim is that concern graphs are *cost-effective*. Our third claim is that concern graphs are *robust* enough to describe concerns across different versions of a system.

To validate these claims, we have performed a series of five case studies of program evolution using concern graphs. Each study was designed to investigate specific research questions, focusing on one of the aforementioned claims. We favored multiple, focused case studies over a large holistic one because we were interested in gathering empirical data about specific aspects of the approach, and because performing multiple studies allows us to determine whether our data is corroborated between studies.

The five case studies we performed involved five different software systems of different style, and of size varying between 12.5 and over 100kLOC. Table I lists the five studies conducted and the claim that is the focus of each study. We

[11]As measured using Eclipse 2.1 on a Windows XP 2002 PC with a 1.8 GHz Intel Celeron processor and 512 MB of RAM.

refer to each study by the name of the system evolved or investigated as part of the study.

We conducted the first three studies (AVID, Jex, and Redback) using an early prototype of FEAT that was developed as a stand-alone Java application. Since these studies are presented in detail elsewhere [Robillard and Murphy 2002], we only provide a summary of each.

The jEdit and ArgoUML studies are two larger-scale studies performed using the FEAT tool described in this article. These studies are presented in detail.

5.1 The AVID Study

The first study was exploratory. In this study, a researcher (one of the authors of this article) took the role of a maintenance programmer to perform a modification to AVID, a Java visualization software system developed at the University of British Columbia [Walker et al. 1998]. AVID comprises 12,853 noncomment, nonblank lines of code organized in 177 classes and 16 packages. The participant for this case study had no previous exposure to the code of AVID. The goal was to assess the practical benefits of using concern graphs. The study consisted of performing a complete change task for AVID using the early FEAT tool prototype. The data collected during the study consists of the modified version of the AVID code, the concern graph produced, and a log of the actions performed in the tool. This study resulted in two main findings.

(1) The concern graph provided an adequate abstraction of the code relevant to the change. Implementation details not captured by the concern graph could easily be found and understood by studying the source code surrounding the projection of a concern graph.
(2) The concern graph provided good support for documenting the list of methods that needed to be changed. It also provided a quick means to perform additional investigation when information was missing.

5.2 The Jex Study

For our second study, we were interested in evaluating whether developers unfamiliar with concern graphs and FEAT would be able to build a concern graph for the code related to a change, without difficulties or extensive effort. In this case study, a subject was asked to use the early FEAT prototype to identify the code contributing to a specified concern in the context of a program change task. We replicated the study three times with three different subjects. The subjects were not asked to perform the change. The target for this task was the Jex system version 1.1 [Robillard 2003b; Robillard and Murphy 1999]. Jex is a static analysis tool that produces a view of the exception flow in a Java program. It is written in Java and consists of 57,152 noncomment, nonblank lines of code organized in 542 classes and 18 packages. The subjects were asked to report the time required to perform the task, their final concern graph, a usage log automatically generated by FEAT, and their confidence in the quality of the result in terms of estimated percentage of the concern code they had missed. We analyzed the completeness of the concern graph produced, and

the usage patterns of the subjects. We also took into account the time taken by each subject to perform the study. For the analysis of completeness, we used the concern graph produced by the author of Jex as a benchmark. The subjects, all of whom had received only minimal training with the FEAT tool, were able to create (in less that 50 minutes and from an unfamiliar code base of 57 kLOC) a concern graph that captured many pertinent parts of the concern. We concluded that building a concern graph during program investigation activities does not require specialized skills, elaborate training, or undue effort.

5.3 The Redback Study

We performed a third study to assess the scalability of our approach. This study consisted of producing a program model of a large industrial code base with the early FEAT prototype, using the tool to capture existing scattered concerns in the code base, and observing and documenting any issues associated with the scalability of the approach. We applied FEAT to the Redback Canada NSC code base. The NSC code base is comprised of 233 packages and 1,489 classes. It depends on approximately 9MB of third-party libraries. The approach taken in the FEAT tool is to load the entire program model into memory. This approach allows users to quickly perform dependency analyses on any parts of a program, and to dynamically reconfigure the environment used to evaluate the queries. In the case of the NSC code base, it was not possible to load all of the application classes and their dependent classes into the memory available on the analysis machine.[12] The very large size of the NSC code base necessitated finding a way to selectively restrict the in-memory model of the program. We accomplished this restriction by modifying FEAT to fully load only a user-defined set of classes. This mechanism was retained as a key aspect of the approach and implemented in our current version of the FEAT plug-in.

5.4 The jEdit Study

The fourth study in our series, and the first to use the FEAT Eclipse plug-in, was intended to strengthen our prior validation of the functionality of concern graphs. This study involved replicated observation of a complete software evolution task involving scattered concerns on a text editor called jEdit. We aimed our investigation at gathering evidence that concern graphs help developers discover and manage knowledge about scattered concerns more effectively than developers not using concern graphs. Since space constraints prevent us from presenting the complete details of the study, this section contains an extended summary. The complete details can be found in a separate report [Robillard 2003c].

5.4.1 *Study Design.* The basic design for this study was to monitor the activities of different subjects performing a complete program evolution task with or without the FEAT tool. Specifically, we replicated the investigation with two subjects using FEAT and two subjects not using FEAT (the control group).

[12]The machine used had 256MB of memory.

We chose to study four subjects to strike a balance between the requirement to study a realistic task in detail and the cost and difficulty of analyzing enough of the data collected to account for the complexity of the phenomenon observed. We chose to contrast the use of FEAT with the use of Eclipse functionality because at this stage in the experiment, we were interested in understanding how the use of FEAT impacts the behavior of developers working with tools representing the state of the practice. For this purpose, Eclipse provides us with a stable, well-known, and well-understood baseline. For a more thorough investigation of the question once the approach matures, it will be necessary for future studies to also consider the differences between users of FEAT and users of different program exploration tools.

The target system for the task was the jEdit text editor (version 4.1-pre6).[13] jEdit is written in Java and the version we used consists of 64,994 noncomment, nonblank lines of source code distributed over 301 classes in 20 packages. Among other features, jEdit saves open file buffers automatically. Our study focuses on this autosave feature. The task presented to the subjects was to modify the autosave feature so as to allow users of jEdit to explicitly disable it through a user interface command.

Understanding the complete set of functionality related to the change task involves reasoning about the use of approximately five fields and 27 methods scattered in 10 classes. The change, as initially performed by the first author of this article in preparation for the study, amounted to about 65 lines scattered in six classes.

The study was divided into the following phases:

—*Eclipse training phase*: We first had the subjects complete a tutorial on how to use the principal features of Eclipse required for the study: code browsing and editing, and performing searches and cross-references. This phase was limited to 30 minutes.

—*FEAT training phase*: The subjects assigned to the FEAT group were required to complete a 90 minute training tutorial on the FEAT tool. At the end of the tutorial, the subjects were tested to ensure that they had successfully met the objectives of the training.

—*Program investigation phase*: After completion of all training, a subject was asked to read some preparatory material about the change to perform. This material included excerpts from the jEdit user manual describing file buffers and the autosave feature, instructions on how to launch jEdit and test the autosave feature, the change requirements, a set of eight test cases covering the basic requirements, and two "seeds" (i.e., pointers to the code) intended to simulate expert knowledge available about the change task. A subject assigned to the FEAT group was given these same seeds in the form of two preloaded concerns in the FEAT tool, each containing one class. A subject was then given one hour to investigate the code pertaining to the change in preparation for the actual task. A subject was to investigate the code using the search and cross-reference features of Eclipse (for the control group) or

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

Table II. Subject Characteristics

Subject	Eclipse	Experience	Java
C1	Low	High	High
C2	Low	Low	High
F1	High	High	Low
F2	High	Low	Low

the queries of the FEAT tool (for the FEAT group). During the program investigation phase, we recorded all of the subjects' activities using the Camtasia screen recording program[14] operating at five frames/second and a resolution of 1,280 × 1,024 pixels.

—*Program change phase*: In this phase, a subject was instructed to implement the requirements as well as possible, and given two hours to implement the change. This phase was also recorded using the Camtasia screen capture program.

5.4.2 *Study Subjects.* In this study, the FEAT subjects are referred to as F1 and F2, and the control subjects as C1 and C2. The four subjects were all experienced programmers. Table II provides a relative evaluation of each subject's characteristics. Data in the Eclipse column indicates the level of proficiency with the Eclipse development environment. A high value indicates that the subject had used Eclipse for real development tasks, whereas a low value indicates that the subject either had never used or had only tried Eclipse. The experience column indicates the overall programming experience of each subject. A low value indicates between three and five years of full-time programming experience (or the equivalent); a high value indicates more than five years of experience. The Java column indicates the experience of each subject with the Java language. A low value indicates less than one year, while a high one indicates between two and three years of experience. When recruiting for the FEAT group, we looked specifically for subjects with a lower level of Java programming experience so that any improved performance compared to the control group could not be correlated with Java coding experience. As a consequence of controlling for programming experience with Java, our control subjects were those two with less Eclipse experience. Section 5.4.4 discusses this factor in more detail.

5.4.3 *Results.* The solution implemented by all four subjects passed all of the test cases provided. In addition, manual inspection of the source code produced by each subject ensured that their solutions respected the existing design of jEdit (as opposed to being "hacks"). The solutions differed only in minor implementation decisions. It is important to note that developer behavior, not implementation quality, is the dependent variable in our analysis; quality is dependent on too many factors to be directly evaluated. As such, we consider that a relatively homogeneous quality of solutions between our subjects, instead of confusing the results, adds to the validity of the study by ensuring the adequacy of skills of the subjects in the study.

[14]http://www.techsmith.com

For a complete investigation of the details of the behavior of each subject to remain tractable, we needed to focus on a subset of the task. For this subset, we chose to study how the subjects approached and handled disabling the recovery mechanism triggered by jEdit when backup files are detected. The correct implementation of this requirement requires discovering and understanding the call between methods `load(View,boolean)` and `recoverAutosave(View)` of class `Buffer`. Method `load(View,boolean)` loads a file buffer in memory. If an autosave file for the buffer is detected, it calls method `recoverAutosave(View)` to perform the recovery. The implementation of the requirement involves testing whether the autosave property is enabled, and if it is disabled, either modifying `load(View,boolean)` to bypass the call to `recoverAutosave(View)`, or modifying `recoverAutosave(View)` to perform no action.

In the context of our analysis, this simple implementation concern presents several desirable characteristics.

—The call to `recoverAutosave(View)` is not located near any code dealing with other issues of the autosave concern. As such, the likelihood of the call being discovered by chance is small; it requires a conscious effort.

—As opposed to other requirements for the task, the call to `recoverAutosave(View)` is not directly related to any member of the two classes given as a seed to the subjects. As such, the information cannot be discovered through a simple query on the initial clues.

—In contrast to other requirements for the task, the subset of code to identify corresponds to a single and precise point in the program. There can exist no ambiguity about whether the subjects have found the right information.

For these reasons, the behavior of subjects investigating and implementing the subset of the task described earlier is likely to be representative of the behavior of subjects in a real setting and provides a good benchmark for comparison.

Our detailed analysis of the behavior of each subject consisted of recording how the call between the two *benchmark methods* `Buffer.load(View, boolean)` and `Buffer.recoverAutosave()` was discovered and the corresponding code modified. All descriptions are based on the screen capture movies collected during the study. The detailed behavior of each subject, and corresponding references to the transcripts of the movies relevant to this analysis, are presented in a separate report [Robillard 2003c].

—*Subject C1.* Subject C1 found `recoverAutosave` accidentally three times before recording the method in a free-form text file. This information was not captured efficiently: The subject made typographical errors while recording the name of the method, requiring multiple window switches. When he retrieved the code of the method at a later stage, the subject needed to browse numerous elements declared in the `Buffer` class to find it, in this case also making a mistake by selecting the wrong element.

—*Subject C2.* Subject C2 examined the benchmark methods multiple times before the link between them (and their relevance to the change task) was identified and recorded explicitly. In particular, methods `load` and `recoverAutosave`

were recorded in the subject's notes in two separate events, separated by code browsing. Subject C2 accessed the `load` method immediately after consulting the notes. Like in the case of C1, accessing the `load` method required browsing irrelevant information in a code browser.

—*Subject F1.* Using FEAT, the subject discovered a relevant method while systematically traversing a section of the control flow relevant to loading file buffers using FEAT's "calls" query. This subject needed to access the benchmark methods twice: once to implement the change, and a second time to fix a bug. Each time, the location in the code was accessed directly through the concern description, avoiding browsing and traversal of irrelevant code.

—*Subject F2.* Subject F2 found the recovery method while systematically investigating the accessors of a field related to autosave functionality. The information was recorded precisely, as the subject captured only the relation between `load` and `recoverAutosave` in a concern, and used only this information when making the change.

—*Interpretation.* Our research question for the jEdit study was to determine how developers use concerns graphs during program evolution, and whether any distinguishing characteristic of this behavior indicates evidence of improved effectiveness. The data from our case study shows that subjects F1 and F2 used concern graphs (as embodied in the FEAT tool) as envisioned. First, they used queries over the program model provided by FEAT to systematically investigate one concern at a time. Second, they recorded precisely the information relevant to implementing the autosave recovery in a distinct concern, and used only this information when actually implementing the change. Contrasting the behavior of FEAT subjects with control subjects, we see that the investigation performed by users of FEAT was more precise. Both FEAT subjects found one of the benchmark methods while investigating structural relations to elements relevant to the implementation of the autosave feature; in contrast, both control subjects found one of the benchmark methods serendipitously, while browsing members of the `Buffer` class. Also, information discovered as part of the investigation was recorded more effectively by FEAT subjects. Screen capture movies show the control subjects recording information about the task by voluntarily writing the name of elements in a textual file, a process requiring multiple view switches, and, in the case of C1, the correction of an error. During the change phase, all four subjects found it useful to access the information they had recorded about interactions between benchmark methods so as to find the location in the code in which to implement the change. Again, the movies for FEAT subjects show a streamlined process when accessing the captured information, with the subjects selecting the relevant concern in the Concern Graph view and accessing one of the benchmark methods. On the other hand, subjects in the control group needed to view their notes, then browse many unrelated elements in a code browser in order to find the benchmark methods. In one case, the subject ended up selecting the wrong element.

5.4.4 *Validity.* To limit threats to the construct validity of the study, our analysis relies on the basic transcripts, and sometimes directly on the actual screen capture movies. This way, minimal divergence is introduced between measures of the subjects' behavior and their actual behavior.

The internal validity of our study is threatened by the possibility that the success level and behavior of a subject is determined by a different, competing factor, such as prior knowledge, proficiency with the development environment, or investigator bias during the study. To reduce this possibility, we took steps to ensure that no subject had prior knowledge of jEdit, asked subjects not to communicate details of the study to others, provided basic training with Eclipse to each subject, precluded the use of powerful features of Eclipse (such as the debugger), and scripted the entire study, limiting the role of the investigator to answering questions. There is always the possibility of investigator bias in the answers to subjects' questions. To limit this effect, we established guidelines at the start of the study for the investigator to use in answering questions: The investigator was to answer questions only about features of the tools covered in the tutorial, and was not to provide any comment about the task. Regarding the impact of the fact that both control subjects had less experience with Eclipse than the FEAT subjects, we believe that this factor is mitigated by our assessment of Eclipse proficiency after the training period, the disabling of powerful Eclipse features, and the fact that the FEAT subjects were new to the FEAT user interface and were thus also operating in a learning context. The internal validity of the results of the study are also threatened by our selection of a specific subset of the task for detailed analysis. To mitigate the influence of this decision on the result, we picked that subset which we considered the least likely to influence the analysis through the experimental procedures.

Finally, a limit on the generalizability of our findings comes from the fact that we have based our observations on the analysis of the behavior of four subjects. We could gain additional evidence supporting our claims by studying more subjects from a larger cross-section of backgrounds. However, studying a small number of subjects has an important benefit: It allows us to analyze unabstracted details of the program investigation behavior of each subject, which greatly increases construct validity.

5.5 The ArgoUML Study

To assess whether concern graphs were robust enough to describe concerns in real evolving software, we performed a study of the evolution of a large system on which a concern graph was defined. As the target application for this study, we chose ArgoUML,[15] a tool for producing diagrams in the unified modeling language (UML) [Robbins et al. 1997]. ArgoUML is developed in Java and totals between 92 and 100kLOC, depending on the version considered. The code base, revision history, and bug database of ArgoUML are publicly available.

5.5.1 *Study Design.* The question we investigated in the ArgoUML study is whether a concern graph can represent the implementation of a concern in

[15]http://argouml.tigris.org

two different versions of a system. For the study, the first author of this article, the study subject, created a concern graph that captured the code related to the correction of bug 1,209 identified for version 0.11.4 of the system and fixed in version 0.13.4. The subject then loaded the concern graph on version 0.13.4 of the system and analyzed the information described by the concern graph. The justification for describing code related to a bug that had been fixed was to ensure that the concern graph would describe code that had changed. In the rest of this section, versions 0.11.4 and 0.13.4 will be referred to as versions 1 and 2, respectively.

The report for bug 1,209, obtained from the bug database for the ArgoUML project, refers to the possibility of attaching a notes (or comments) box to different objects in UML diagrams. In version 1, it is only possible to attach notes to objects in class, state, or activity diagrams. Fixing this bug requires modifying the code of ArgoUML to support adding notes to all diagram types.

5.5.2 *Results.* Without looking at the code of version 2, the subject created a concern graph capturing code which seemed relevant to the evolution task. The resulting concern graph was comprised of 41 fragments, totaling three fields, 33 methods, and 26 classes (including eight library classes), scattered in 16 different packages. The concern graph captured code related to the creation of a new note element in the internal UML model, the user interface code supporting creation of a new note, code to add the notes button to the toolbar, classes implementing the different UML diagrams, and the code to display note objects on a diagram. The subject created the concern graph by performing queries with the FEAT tool.

Version 2 (0.13.4) of ArgoUML is not the direct successor of version 1 (0.11.4). To test our approach as thoroughly as possible, the second version we chose was instead a much later revision, released close to six months after version 1. As such, it implements a great number of changes, including the fix to bug 1,209. It is approximately 8kLOC larger than version 1. To measure how much of the code of ArgoUML relevant to our concern had actually changed between versions 1 and 2, we used the code comparison feature of Eclipse, which works in a way similar to the UNIX `diff` utility [Hunt and Szymanski 1977]. For each of the 18 nonlibrary classes of version 2 in the concern graph, we calculated the number of lines in text sequences that did not match version 1, and divided this by the total number of lines in each class declaration in version 2 to produce an approximate relative metric of change. The results show that all but one of the nonlibrary classes in the concern graph had changed, with the ratio of change varying between 2% and 86%. The average change was 51%.[16]

After loading the concern graph on version 2, accessing the Inconsistency View (see Section 4.3.1) revealed that seven fragments were inconsistent (out of 41).

The first inconsistent fragment we considered is the primitive fragment with intension (and extension):

`UMLActivityDiagram.initToolBar()` `calling` `ToolBar.add(Action)`

[16]The complete details of this assessment can be found in Robillard [2003c].

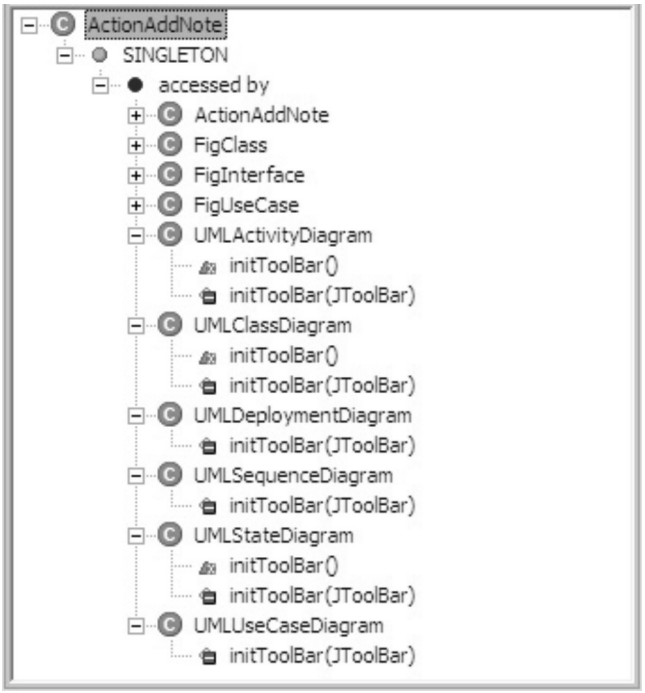

Fig. 10. Representation of an inconsistent fragment in the FEAT Inconsistency View.

Selecting this inconsistency in the Inconsistency View revealed that method `initToolBar()` of class `UMLActivityDiagram` no longer existed. By performing two FEAT queries based on this inconsistent fragment, we could establish that the method `initToolBar()` had been moved from class `UMLActivityDiagram` to its superclass `UMLDiagram` in the new version. The first query revealed all the callers of `add(Action)`, which included a call to a method `initToolBar()` in class `UMLDiagram`. The second query verified that `UMLDiagram` was the superclass of `UMLActivityDiagram`. To repair this inconsistency, we replaced the inconsistent fragment with the call to `ToolBar.add(Action)` from `UMLDiagram.initToolBar()`.

The next five inconsistencies we considered were also caused by the move of method `initToolBar()` to the superclass. These inconsistencies were detected and repaired using the same approach as that detailed previously.

The last inconsistency was a fragment with a universal range:

`ActionAddNote.SINGLETON accessed by ALL`

Displaying this inconsistent fragment in the Inconsistency View revealed that the inconsistency was caused by changes to a method named `initToolBar` in six different classes. Figure 10 shows the preceding fragment as it appears in the fragment viewer of the FEAT Inconsistency View.

Based on annotations in the figure and by drilling down to the source code, we determined that method `initToolBar` was changed to include a parameter in

```
(a) ((MBase)o).addMElementListener(
          UmlModelListener.getInstance());

(b) ((MBase)o).addMElementListener(
          UmlModelEventPump.getPump());
(c) ((MBase)o).addMElementListener(
          UmlModelListener.getInstance());
```

Fig. 11. Source code mapping for a same fragment in ArgoUML versions 1 and 2.

classes `UMLActivityDiagram`, `UMLClassDiagram`, and `UMLStateDiagram`, and that an access to field `SINGLETON` of class `ActionAddNote` was added in the three other classes. This modification clearly corresponds to the bug fix identified before, as the three classes missing the field access represent diagram types for which note objects were not initially supported.

To fix these inconsistencies, we used the automatic repair feature of FEAT, and synchronized the concern graph with the code according to the repair function (Definition 12). The complete process of investigating and repairing this inconsistency took only a few minutes, which we believe to be much shorter than it would have taken to rediscover the same information through traditional code searches.

To complete this case study, we show an example of a fragment that remained consistent in the face of extensive change to a method, and which allowed us to view different source code mapping to the same intension. The concern graph contained the fragment with intension (and extension):

`AbstractUmlModelFactory.initialize(Object)` `calling`
`MBase.addMElementListener(MElementListener)`

The source code of method `AbstractUmlModelFactory.initialize(Object)` in version 1 corresponding to the fragment is shown in Figure 11(a).

The source code for the same fragment in version 2 corresponds to statements 11(b) and (c). Since both of these statements are still in method `AbstractUmlModelFactory.initialize(Object)`, the projection remains unchanged. As this example shows, capturing the essence of a concern in terms of structural dependencies allows us to preserve the intension of a concern in the face of changing source code.

The case study has shown how a concern graph can be used to describe the same concerns in multiple versions of a code base, and provides evidence to support the claim that concern graphs can have value in more than one version of a system. More specifically:

—Basic tolerance to minor changes in the code allows some information to be reused as-is. In our case, 34 out of the 41 fragments analyzed (83%) remained consistent. In other words, a developer who would need to understand how the notes feature is implemented in ArgoUML could use 83% of the information originally captured at no additional cost. This tolerance

is possible because the concern graph model abstracts details of the source code.

—The concern graph model and tool support are enough to detect inconsistent information, and thus prevent users from unknowingly using this information. Because both the intension and extension of fragments are stored in a concern graph, it is possible to detect not only that subset of a concern graph which is invalid, but also the mismatches between an extension and the projection of its intension. This support provides more than the dependency analysis provided by compilers: It allows reasoning about structural differences between versions that are separated by multiple releases.

—In cases where it is desirable, it is possible to repair inconsistencies between a concern graph and a program model. Because a concern graph consists of a set of atomic building blocks (fragments), it is possible for a developer to understand the inconsistencies between a concern graph and a system in a very specific context, making it easier to repair the inconsistencies. Also, even though repairing inconsistencies entails additional effort from a developer, this effort is minimal (e.g., performing two queries) in contrast to rediscovering the equivalent information through traditional means.

5.5.3 *Validity.* The main threats to the validity of the ArgoUML study are investigator bias in choosing the concern to create, and the creation of the concern graph. To mitigate these factors, we used a modification of ArgoUML related to an actual bug as a case, as opposed to investigating an arbitrary concern in the code base. Second, the concern to investigate was selected before any investigation of the source code was performed. It was thus not possible for the investigator to select a concern that would have a good chance of evolving well. Additionally, the source code for the concern in version 0.13.4 was not examined until the concern graph on version 0.11.4 was completed. As such, it was impossible for the investigator to know in advance how stable the concern graph would be. Construct validity is not an issue in this study, since no surrogate measure is used.

5.6 Synthesis of the Validation

Taken separately, each study presents an incomplete picture of the use of concern graphs. In each case, we have made concessions to the necessities of practical empirical investigation involving a prototypical tool. In choosing to validate our approach using multiple case studies, our goal was to perform an in-depth assessment of key aspects of the approach. In addition, the data collected from one study often corroborates and strengthens claims that were the focus of another. For example, the jEdit study, which was designed to assess the claim that concern graphs are functional, also showed that subjects using FEAT did not expend any significant effort building a concern graph. This observation also increases the evidence supporting the claim that concern graphs are cost-effective. In brief, the validation studies presented in this section have the advantage of representing repeated experiences with the approach in different

circumstances, and with different developers performing different tasks on different systems. The overlap of data validating the different claims, the lack of obvious contradictions between studies, and the variety of systems and tasks studied contributes to the generalizability of our results in similar circumstances. Finally, reports on the use of concern graphs to support other research projects [Hannemann et al. 2003; Murphy et al. 2004; Souter et al. 2003] are additional indications of the value of the approach.

6. DISCUSSION

Interesting issues and questions arose during the development and validation of the concern graphs approach that open the door to additional investigation.

6.1 Training and the Use of FEAT

One important factor influencing the adoption of new software development approaches is the effort of training potential users. As part of our empirical studies, we observed that the effectiveness of the training provided to FEAT users is influenced by three overlapping factors: prior exposure to the concept of separation of concerns, experience with program analysis and cross-reference queries, and experience with the use of software development environments. For the jEdit study, after prototyping the study with a FEAT training session of 60 minutes, we determined that this amount of time was insufficient, and increased the training time to 90 minutes for the final study. After this amount of training, both FEAT subjects involved in the jEdit study were able to use the tool properly. A training time of 90 minutes thus constitutes a lower bound on the effort required to use the FEAT tool effectively.

6.2 Capturing System Behavior With Concern Graphs

When analyzing the results of the AVID and jEdit studies, we observed that the code relevant to a concern sometimes included complex program behavior (e.g., constraints on the order of calls to a method). The program model extracted by FEAT does not support the investigation and capture of this kind of behavioral information. This observation raises two important questions. First, can concern graphs provide any help for complex cases? Second, should more support be provided? In answer to the first question, case studies have shown that concern graphs are helpful to developers because they provide a means to store a set of program elements that can act as anchors and provide context when investigating complex code. In other words, although concern graphs cannot explicitly capture complex interactions, they can point to those places where such interactions occur, and, through concern names, provide some information about the context in which they occur. Evidence of this type of support is found in both the AVID and jEdit studies. For example, in the jEdit study, both subjects, having realized that some part of a concern was not well understood, used the concern graph to return precisely to the point where further investigation was required. Concern graphs thus provide some minimal support for understanding complex code. Although changes to the model and supporting technology can be made to support richer concern graphs, these alternatives have important associated

costs. Determining whether finer-grained concern graphs provide a more desirable cost-benefit tradeoff remains an open question.

6.3 The Importance of a Good Seed

In our description of the concern graph approach, we have assumed that a developer knows a relevant program location from which investigation can start. Based on such a starting point (or seed), a developer can investigate related elements in the source code and build a concern graph. Because concern graphs are designed to support a very focused investigation of the source code, the approach is not intended to assist with the broad type of investigation related to identification of a seed. The identification of a seed is a separate phase of a program maintenance task, performed outside of the FEAT tool. There exists a variety of ways in which a developer can obtain a seed for the investigation of code pertaining to a maintenance task. We can, for example, rely on other developers. This is the approach we have used in the AVID study, and have simulated in the jEdit study. Other possibilities include broad lexical searches for relevant keywords in all the source files, use of clues in the user interface of a system [Michail 2002], and specialized feature location techniques [Eisenbarth et al. 2003; Wilde and Scully 1995; Zhao et al. 2004].

During the evaluation of a technique for automatically inferring concern code from program investigation activities [Robillard and Murphy 2003], we observed that developers unfamiliar with a code base performed much more focused and effective program investigation if a good seed had been provided. This observation has two important consequences for potential adopters of the concern graph approach. First, we should only attempt to build a concern graph once a relevant seed has been identified; failing to do so may result in wasted effort documenting irrelevant information. Second, and more importantly, a database of concern graphs can provide an alternative source of potential seeds for other program evolution tasks. By perusing the concerns other developers have built for tasks similar to a task at hand, a developer can potentially discover a good seed.

6.4 Concern Interaction Analysis

One of the characteristics of the general concern graph model is the support for analyzing interactions between two concerns (Sections 3.4 and 4.3). In devising and implementing support for concern interaction analysis, our goal was to increase the usefulness of concern graphs by providing a means for developers to analyze whether and how two concerns interact, without having to peruse the entire concern descriptions and perform the analysis mentally. We informally evaluated the contribution of the concern interaction analysis to the usefulness of the concern graph approach as part of the jEdit study. In this study, the training documentation for users of FEAT included detailed information about how to use concern interaction analysis, examples for subjects to practice on using the feature, and instructions detailing the situations when concern interaction analysis could be useful. In spite of these provisions, neither of the subjects who performed the evolution task on jEdit with FEAT used concern

interaction analysis in more than a cursory and exploratory way. By interviewing the subjects, we established that they had not used interaction analysis because it had not been deemed useful. Specifically, having just built a concern graph, the information captured by the concern graph was still fresh in the subjects' memory, and the concern interaction analysis was not seen as providing significant help for the task. The usefulness of the concern interaction analysis thus remains to be formally evaluated. Our hypothesis is that although it does not seem to be useful for developers initially investigating a concern, it may help other developers accessing the concern at a later stage.

7. RELATED WORK

Early empirical evidence that scattered concerns pose problems to programmers was collected by Soloway et al. during different studies of professional programmers [Letovsky and Soloway 1986; Soloway et al. 1988]. In one study conducted at NASA's Jet Propulsion Laboratory, Soloway et al. [1988] observed that the programmers who did not implement a correct modification to a small system "failed to understand the casual interactions inherent in one of the key delocalized plans" [Soloway et al. 1988, p. 1262]. To address the difficulty of performing maintenance on code involving delocalized plans (in other words, scattered concerns), the researchers propose that programmers produce explicit documentation detailing delocalized plans in programs. Their initial approach is a form of paper documentation, where source code is presented in parallel with pointers linking the code to other relevant sections of a program, and detailing the rationale for different design and implementation decisions. Although the idea of Soloway et al. [1988] is based on sound empirical observations, free-form documentation is effort-intensive to produce and maintain. Since this early work on delocalized plans, a number of approaches have been proposed to document the implementation of scattered concerns in software systems. In the rest of this section, we describe the body of work in software engineering whose explicit focus is the representation of concerns in source code.

Descriptions of scattered concerns can be captured as *virtual files*. In software development environments, the idea of a virtual file is to present various segments of source code and other system documentation relevant to a task as a single unit. For example, in the Desert environment [Reiss 1996], a developer can load, edit, and save a virtual file consisting of fragments of other source files. Stellation [Chu-Caroll et al. 2002; Chu-Carroll and Spenkle 2000] is a fine-grained software configuration management system that supports method-level storage management. Using a concept similar to Desert, Stellation supports virtual source files using a typed aggregation mechanism that collects different program elements and other artifacts (such as test cases) in a single unit for the purpose of configuration management. The proposal for Stellation includes the possibility of specifying aggregates either explicitly or as the result of a query. Virtual files could be used to document scattered code that implements a concern. However, to compose a virtual file, a developer must already know about the location of the code implementing a concern. The mechanisms proposed for Desert and Stellation also do not include support for

tolerating and managing inconsistencies between a virtual file and the source code.

A number of approaches support the representation of concerns based on characteristics of the program text forming a software system. Conceptual modules [Baniassad and Murphy 1998] allow developers to specify concerns as a collection of scattered lines of source code, and to perform analyses on the resulting modules. Because conceptual modules are defined in terms of lines of source code, they cannot be reused with different versions of a system. The Aspect Browser is a tool proposed to help developers find concerns using lexical searches of the program text [Griswold et al. 2001]. Concerns found in this fashion can be stored and viewed at different times to support program evolution tasks. Aspect Browser uses the Seesoft [Eick et al. 1992] concept and a map metaphor to graphically represent the location of code implementing concerns in the context of the entire code base. The aspect mining tool (AMT) [Hannemann and Kiczales 2001] is conceptually similar to Aspect Browser, but supports additional queries based on types. The Aspect Browser and AMT can be used both for finding and documenting concerns. However, because they only support the specification of concerns based on lexical matches to regular expressions and the use of types, their expressive power is limited. The text-oriented approach also limits the tools' ability to capture relationships between scattered program elements explicitly.

Other approaches support the representation of concerns in source code based on structural properties in ways that provide cost-benefit trade-offs that are different from concern graphs. Intentional Views [Mens et al. 2002; 2003] 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 thus offer more flexibility, but at a higher cost to developers, who need to explicitly form the queries describing the views. Intentional Views are also generative and do not store the subset of a program generated from a view, so they do not allow the type of reasoning about changes to the source code that concern graphs support. Cosmos [Sutton and Rouvellou 2002] is an approach that has been proposed to model concerns in terms of high-level system characteristics. As opposed to concern graphs, the Cosmos model currently does not support maintaining links between concern descriptions and the corresponding source code. The concern manipulation environment (CME) [Harrison et al. 2004] is a project whose goal is to integrate support for creating and evolving aspect-oriented software across the life cycle of a system. CME includes a concern explorer tool that can be used to describe concerns using queries in a way similar to FEAT. FEAT and CME followed parallel research courses and share many similarities. However, the research focus for each project is different. While CME is aimed at providing a unified way to represent concerns across different types of software engineering artifacts, FEAT focuses on linking concern descriptions and other artifacts with the source code. As a result, CME is more open (supporting, e.g., relations between elements not found in the code), while FEAT supports a higher level of reasoning about the links between concerns and source code (e.g., concern interactions expressed as concerns, and

analysis of inconsistencies between a concern description and a version of the source code).

8. CONCLUSIONS

Evolving a software system typically requires a developer to locate and understand concerns that are not adequately separated in the code. To help developers cope with software change tasks involving scattered concerns, we propose to explicitly model the implementation of concerns in software engineering artifacts. We developed a formal representation for concerns in source code, called concern graphs. Concern graphs are an abstraction of the implementation of a concern that can be mapped onto the actual source code of a system.

To experiment with concern graphs, we have developed a tool called FEAT, which allows developers to iteratively build concern descriptions as the source is investigated, to view the code related to a concern, and to perform analyses on the concern representation. Concern graphs also support a specialized mechanism, implemented in FEAT, that enables the detection, management, and repair of inconsistencies between a concern graph and an actual code base.

Using FEAT, we have evaluated the costs and benefits of concern graphs in a series of case studies involving the evolution of five different systems of differing size and style. The results show that concern graphs are inexpensive to create during program investigation, support views and operations that facilitate the task of modifying the code implementing scattered concerns, and are robust enough to be used with different versions of a system.

In conclusion, the concern graph approach is an integrated and practical solution to the problems of finding, viewing, and preserving information about the implementation of concerns in source code.

ACKNOWLEDGMENTS

The authors are grateful to Alex Brodsky, Will Evans, Raymond Ng, Prakash Panangaden, Barbara Ryder, and the anonymous reviewers for numerous and valuable comments on different versions of this article. The authors also thank Jason Xu, whose work on the FEAT plug-in greatly facilitated the tool-related aspects of this work.

REFERENCES

BALZER, R. 1991. Tolerating inconsistency. In *Proceedings of the 13th International Conference on Software Engineering*. 158–165.

BANIASSAD, E. L. AND MURPHY, G. C. 1998. Conceptual module querying for software reengineering. In *Proceedings of the 20th International Conference on Software Engineering*. 64–73.

BELADY, L. A. AND LEHMAN, M. M. 1976. A model of large program development. *IBM Syst. J. 15*, 3, 225–252.

BOEHM, B. W. 1976. Software engineering. *IEEE Trans. Comput. 12*, 25, 1226–1242.

CHEN, Y.-F., NISHIMOTO, M. Y., AND RAMAMOORTHY, C. 1990. The C information abstraction system. *IEEE Trans. Softw. Eng. 16*, 3, 325–334.

CHIKOFSKY, E. J. AND CROSS II, J. H. 1990. Reverse engineering and design recovery: A taxonomy. *IEEE Softw. 7*, 1, 13–17.

CHU-CAROLL, M., WRIGHT, J., AND SHIELDS, D. 2002. Supporting aggregation in fine grained software configuration management. In *Proceedings of the 10th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*. 99–108.

CHU-CARROLL, M. C. AND SPENKLE, S. 2000. Coven: Brewing better collaboration through software configuration management. In *Proceedings of the 8th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*. 88–97.

EICK, S. G., GRAVES, T. L., KARR, A. F., MARRON, J., AND MOCKUS, A. 2001. Does code decay? Assessing the evidence from change management data. *IEEE Trans. Softw. Eng. 27*, 1, 1–12.

EICK, S. G., STEFFEN, J. L., AND SUMMER, JR., E. E. 1992. Seesoft—A tool for visualizing line oriented software statistics. *IEEE Trans. Softw. Eng. 18*, 11, 957–968.

EISENBARTH, T., KOSCHKE, R., AND SIMON, D. 2003. Locating features in source code. *IEEE Trans. Softw. Eng. 29*, 3, 210–224.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns—Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley Longman, Reading, MA.

GOLDBERG, A. 1984. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, Reading, MA.

GRISWOLD, W. G., YUAN, J. J., AND KATO, Y. 2001. Exploiting the map metaphor in a tool for software evolution. In *Proceedings of the 23rd International Conference on Software Engineering*. 265–274.

HANNEMANN, J., FRITZ, T., AND MURPHY, G. C. 2003. Refactoring to aspects: An interactive approach. In *Proceedings of the OOPSLA Eclipse Technology Exchange*. 74–78.

HANNEMANN, J. AND KICZALES, G. 2001. Overcoming the prevalent decomposition in legacy code. Position paper for *the ICSE Workshop on Advanced Separation of Concerns in Software Engineering*.

HARRISON, W., OSSHER, H., SUTTON JR., S., AND TARR, P. 2004. Concern modeling in the concern manipulation environment. Tech. Rep. RC23344, IBM Research.

HUNT, J. W. AND SZYMANSKI, T. G. 1977. A fast algorithm for computing longest common subsequences. *Commun. ACM 20*, 5, 350–353.

KICZALES, G., LAMPING, J., MENHDHEKAR, A., MAEDA, C., LOPES, C., LOINGTIER, J.-M., AND IRWIN, J. 1997. Aspect-Oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*. Lecture Notes in Computer Science, vol. 1241. Springer Verlag. 220–242.

LETOVSKY, S. AND SOLOWAY, E. 1985. Strategies for documenting delocalized plans. In *Proceedings of the Conference on Software Maintenance*. 144–151.

LETOVSKY, S. AND SOLOWAY, E. 1986. Delocalized plans and program comprehension. *IEEE Softw. 3*, 3, 41–49.

MENS, K., MENS, T., AND WERMELINGER, M. 2002. Maintaining software through intentional source-code views. In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*. 289–296.

MENS, K., POLL, B., AND GONZÁLEZ, S. 2003. Using intentional source-code views to aid software maintenance. In *Proceedings of the International Conference on Software Maintenance*. 169–178.

MICHAIL, A. 2002. Browsing and searching source code of applications written using a GUI framework. In *Proceedings of the 24th International Conference on Software Engineering*. 327–337.

MURPHY, G. C., GRISWOLD, W. G., ROBILLARD, M. P., HANNEMANN, J., AND LEONG, W. 2004. Design recommendations for concern elaboration tools. In *Aspect-Oriented Software Development*, T. Elrad et al., eds. Addison-Wesley Longman, Reading, MA USA. 507–530.

OBJECT TECHNOLOGY INTERNATIONAL, INC. 2001. Eclipse platform technical overview. White paper.

O'BRIEN, P. D., HALBERT, D. C., AND KILIAN, M. F. 1987. The Trellis programming environment. In *Proceedings of the Conference on Object-Oriented Programming, Systems, and Applications*. 91–102.

PARNAS, D. L. 1972. On the criteria to be used in decomposing systems into modules. *Commun. ACM 15*, 12, 1053–1058.

PARNAS, D. L. 1994. Sofware aging. In *Proceedings of the 16th International Conference on Software Engineering*. 279–287.

REISS, S. P. 1996. Simplifying data integration: The design of the Desert software development environment. In *Proceedings of the 18th International Conference on Software Engineering*. 398–407.

ROBBINS, J. E., HILBERT, D. M., AND REDMILES, D. F. 1997. Argo: A design environment for evolving software architectures. In *Proceedings of the 19th International Conference on Software Engineering*. 600–601.

ROBILLARD, M. P. 2003a. FEAT: An Eclipse plug-in for locating, describing, and analyzing concerns in source code. http://www.cs.ubc.ca/labs/spl/projects/feat.

ROBILLARD, M. P. 2003b. The Jex home page. http://www.cs.ubc.ca/~mrobilla/jex.

ROBILLARD, M. P. 2003c. Representing concerns in source code. Ph.D. thesis, Department of Computer Science, University of British Columbia, Canada.

ROBILLARD, M. P., COELHO, W., AND MURPHY, G. C. 2004. How effective developers investigate source code: An exploratory study. *IEEE Trans. Softw. Eng. 30*, 12, 889–903.

ROBILLARD, M. P. AND MURPHY, G. C. 1999. Analyzing exception flow in Java programs. In *Proceedings of the 7th European Software Engineering Conference and 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*. Lecture Notes in Computer Science, vol. 1687. Springer Verlag 322–337.

ROBILLARD, M. P. AND MURPHY, G. C. 2002. Concern graphs: Finding and describing concerns using structural program dependencies. In *Proceedings of the 24th International Conference on Software Engineering*. 406–416.

ROBILLARD, M. P. AND MURPHY, G. C. 2003. Automatically inferring concern code from program investigation activities. In *Proceedings of the 18th International Conference on Automated Software Engineering*. 225–234.

SANELLA, M. 1983. *The Interlisp-D Reference Manual*. Xerox Corporation, Palo Alto, CA.

SCHMIDT, G. AND STRÖHLEIN, T. 1993. *Relations and Graphs—Discrete Mathematics for Computer Scientists*. EATCS Monographs on Theoretical Computer Science. Springer Verlag.

SOLOWAY, E., PINTO, J., LETOVSKY, S., LITTMAN, D., AND LAMPERT, R. 1988. Designing documentation to compensate for delocalized plans. *Commun. ACM 31*, 11, 1259–1267.

SOUTER, A. L., SHEPHERD, D., AND POLLOCK, L. L. 2003. Testing with respect to concerns. In *Proceedings of the International Conference on Software Maintenance*.

SUTTON JR., S. M. AND ROUVELLOU, I. 2002. Modeling of software concerns in Cosmos. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development*. 127–133.

TARR, P., OSSHER, H., HARRISON, W., AND SUTTON, S. M. 1999. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering*. 107–119.

VAN GURB, J. AND BOSCH, J. 2001. Design erosion: Problems and causes. *J. Syst. Softw. 61*, 105–119.

WALKER, R. J., MURPHY, G. C., FREEMAN-BENSON, B., WRIGHT, D., SWANSON, D., AND ISAAK, J. 1998. Visualizing dynamic software system information through high-level models. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*. 271–283.

WILDE, N., GOMEZ, J. A., GUST, T., AND STRASBURG, D. 1992. Locating user functionality in old code. In *Proceedings of the Conference on Software Maintenance*. 200–205.

WILDE, N. AND SCULLY, M. C. 1995. Software reconnaissance: Mapping program features to code. *Softw. Maintenance: Res. Pract. 7*, 49–62.

ZHAO, W., ZHANG, L., LIU, Y., SUN, J., AND YANG, F. 2004. SNIAFL: Towards a static non-interactive approach to feature location. In *Proceedings of the 26th International Conference on Software Engineering*. 293–303.

Received February 2005; revised December 2005; accepted January 2006