A position paper for the OOPSLA 2002 Workshop on Tool Support for Aspect Oriented Software Development

Capturing Concern Descriptions During Program Navigation

Martin P. Robillard and Gail C. Murphy
Department of Computer Science
University of British Columbia
{mrobilla,murphy}@cs.ubc.ca

1 Introduction

> On Monday morning, Pat gets assigned the task of restructuring the data model of the company's SuperTool™ software, to optimize performance for speed. Pat opens the software development environment, checks out the 1357 source files of SuperTool, and confidently begins the task. Although not familiar with SuperTool's code base, Pat infallibly knows which file to open, in which order, and what change to apply to each file. At the end of the day, the change, spanning parts of 23 different files, passes the regression tests without a glitch. Pat is hailed as a hero and promoted on the spot.

Although this scenario describes a highly desirable situation, its realism may be challenged. Knowledge of technical implementation details of emergent concerns in software usually exists neither in programmers' heads, nor in technical manuals. Instead, it has to be obtained prior to performing a change to a program. This is usually done by navigating source code, possibly through the use of tools, such as grep [1], or through the cross-reference functionalities of integrated development environments. Unfortunately, the task of program navigation cannot be equated with a search for the mystical "core" of a concern, around which dependent changes radiate. Because, in part, concerns in software are usually scattered and tangled [6], the search is more likely to take the form of a puzzle-solving activity, where different elements in a program are pieced together to form a sufficient understanding of a subset of the program's operations.

This observation leads to the questions: *since we have to navigate programs in any case when dealing with a concern, can we leverage some of the program navigation effort to produce a description of a concern? How? What does this give us?*

The goal of our research is, in part, to answer these questions. This position paper provides a brief summary of some of the issues we are currently considering, (Section 2), outlines a theory serving as the basis for investigating these questions (Section 3) , and describes the implementation of a prototype tool for capturing concerns during program navigation (Section 4).

2 Important Questions

Concern Interfaces Assuming it is useful to capture concerns as entities, what should concern representations look like, at the implementation level? Should concerns be completely separated from the code base, as special modules, like aspects in AspectJ [3]?, or should they be fully integrated with the base code, and indicated through visualization (such a highlighting lines of source code representing a concern [2]). The former approach allows users of the representation to easily abstract the concern from the rest of the base code and treat it as a unit, while the latter can provide important contextual information about the structure and style of the concern's implementation.

Concern Building Blocks If we carry through with the puzzle solving metaphor for building concern representations, then what should the pieces of the puzzle be? And how should they fit together? We can decompose programs at different levels of granularity, from high-level architectural constructs (such as Java packages), down to fine-grained technical concepts such as language expressions and statements used by slicers [7] and other program analysis tools. Existing relationships between elements in a program can be established easily through querying tools. But what is the best level of granularity for cost-effectively describing concerns? In a previous article [5], we argue that statements are too low-level and propose a system based on relationships between global program elements (such as classes and class members in Java). However, the level of granularity to use is not the only question when describing concerns through queries. For example, should transitive or nested queries be allowed to describe a concern, or are they too conceptually difficult to map for the user? Previous user studies showed us that transitive queries were almost never used because of the difficulty in interpreting the results.

Payoff The last question we pose in this position paper is certainly not the least: what should users expect from a concern representation? At the very least, the representation should be an accurate description of the location of the concern in the code, to the level of granularity established as an answer to the previous question. But can we hope for more? We are currently investigating how to support additional reasoning about a concern based on concern representations. Examples include providing the rationale for the inclusion of an element in a concern, and describing the relationships between different concerns.

3 A Theory of Concern Description

To structure our investigation of how various program relationships could be used in a description of a concern, we have devised a framework for the construction of concern representations, based on relational algebra. We summarize this framework here.

A concern representation consists in a set of atomic building blocks called *fragments*. A fragment consists in a *domain*, a *relation*, and a *range*. An example of a fragment would thus be:

```
method1 calls method2
```

In this case, the domain and ranges are methods, and the relation is the method call. Any element defined in a program (classes, methods, fields, local variables, etc.) can be used in this framework as a domain or range, and any relation between the latter can represent a relation in the framework.

A domain or range can be either *primitive*, and consist of a single *element* (as in the example above), *compound*, and consist of a fragment, or *universal* and consist of all applicable elements. Compounds and universal domains and ranges allow the modeling of nested queries, such as "all the methods of `class1` accessing `field1`".

Using this framework, we can experiment with various elements and relations to construct concern descriptions in a standard fashion, and try different heuristics for reasoning about concerns, which are independent of the reification of the framework for a specific programming language or a specific selection of elements and relations.

4 Overview of the FEAT Tool

To experiment with the ideas presented in this paper, we have evolved our earlier concern description tool, FEAT [5], into a plugin (FEAT-2E) for the Eclipse Platform [4], an integrated software development environment with a plugin architecture supporting the addition of functionality (Figure 1, last page).

With the FEAT plugin activated, users of Eclipse can use the integrated development environment as usual, to browse and modify code, perform searches, etc. However, if a user desires to create a concern representation, the FEAT Perspective can be activated and a concern representation created. Any class, method, or field in a project can then be moved to the FEAT Perspective, where it appears in the Projection View (top right window).

Any element in the Projection View can be moved to the Participants View (top center window). The Participant View groups all elements identified as participating in a concern. Elements in either views can be queried for relationships to other elements in the project. Results appear in the Projection View.

When an element from the Projection View is added to the concern, the query used to find the element is also transparently stored as part of the concern (as a fragment). The user can then query any element in the Participants View for its *context*. This results in listing all the relations in which the element is involved which were added by the user.

Users can also view the source code for any element or query. The code corresponding to any query (for example, accesses to a field) is presented in a code viewer and highlighted (bottom window).

Revisiting the questions of Section 2, we can discuss how FEAT represents concerns, what the concern building blocks are, and what we can hope to derive from concern representations.

FEAT represents a concern in a way identical to the base code in Eclipse (i.e., in a declaration tree), except that the tree contains only the elements pertaining to the concern. FEAT also allows to view the concern in the context of the source code.

Relations in FEAT can be any relations between classes and class members. When a query is performed in FEAT, it is expressed as a fragment (this is transparent to the user). If the user adds any result from the query to the concern, the associated fragment gets added to the concern description at the same time. We believe these underlying fragments are the key to providing value-added reasoning power to the concern descriptions. This is how the tool supports querying the context for an element. Using similar strategies, we plan to add views to show the interactions between different concerns.

References

[1] Alfred V. Aho. Pattern matching in strings. In Ronald V. Book, editor, *Formal Language Theory: Perspectives and Open Problems*, pages 325–347, New York, 1980. Academic Press.

[2] William G. Griswold, Jimmy J. Yuan, and Yoshikiyo Kato. Exploiting the map metaphor in a tool for software evolution. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 265–274. ACM, May 2001.

[3] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):51–57, October 2001.

[4] Object Technology International, Inc. Eclipse platform technical overview. White Paper, July 2001.

[5] Martin P. Robillard and Gail C. Murphy. Concern Graphs: Finding and describing concerns using structural program dependencies. In *Proceedings of the 24th International Conference on Software Engineering*, May 2002.

[6] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton. N degress of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering*, pages 107–119. ACM, May 1999.

[7] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

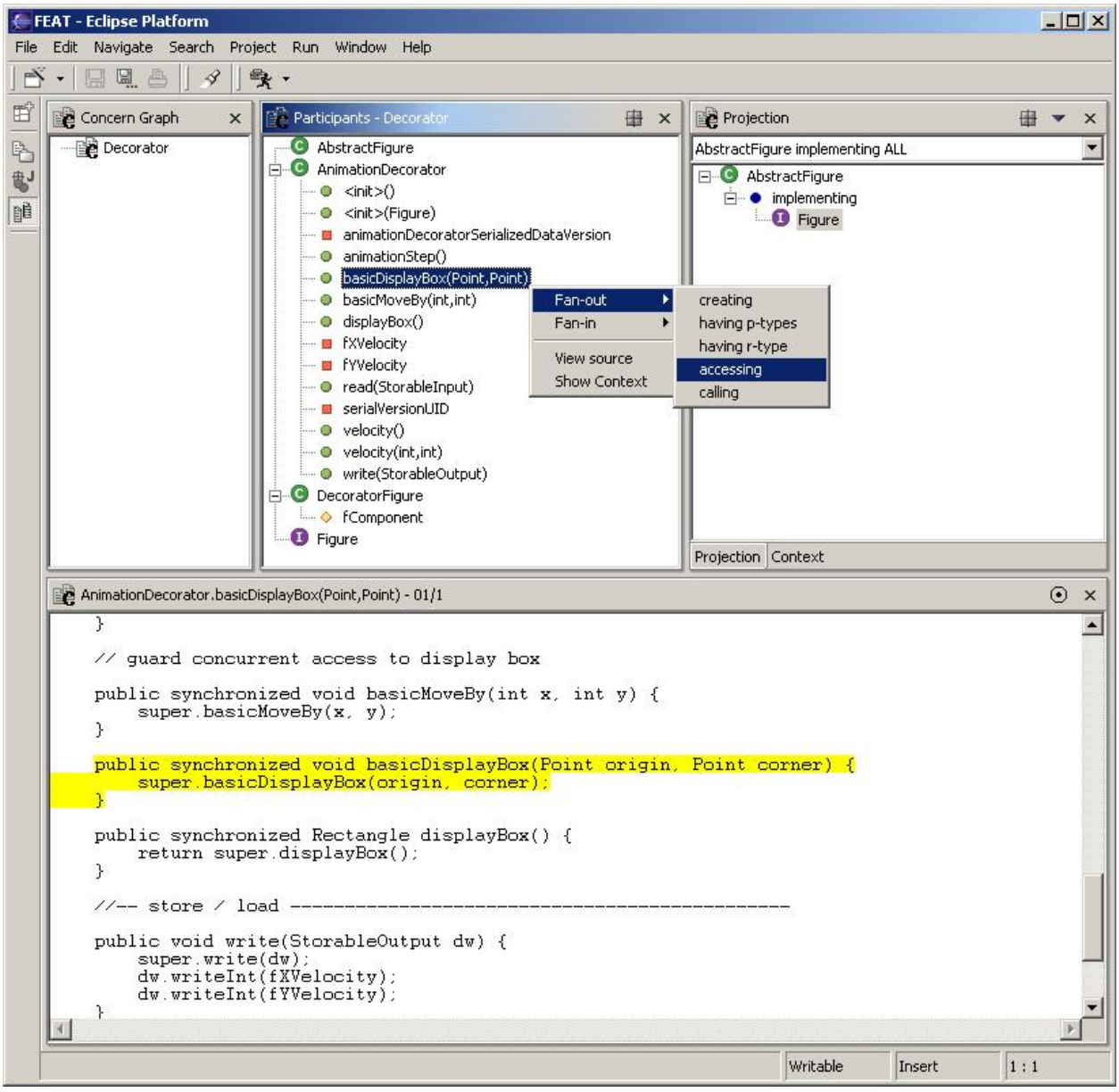

Figure 1: The FEAT tool.