Determining the "Why" of Concerns

Elisa L.A. Baniassad and Gail C. Murphy
Department of Computer Science
University of British Columbia
201-2366 Main Mall
Vancouver B.C. Canada V6T 1Z4
{bani, murphy}@cs.ubc.ca

Christa Schwanninger
Siemens AG, CT SE 2
Otto-Hahn-Ring 6, 81739
Munich Germany
christa.schwanninger@mchp.siemens.de

In a perfect world, a software developer making a change to a software system would fully understand the codebase of the system. Alas, the world is not perfect. It is not possible for most mortal software developers to learn, understand and remember codebases comprising thousands or even millions of lines of code, most of which they did not write. In a growing number of cases, the developer may not have even been born when the code was written.

The structure of existing systems tends to further complicate the developer's task. After some work, a developer may be able to determine which parts of an existing codebase must be altered to affect the desired change. Hopefully, for the developer's sake, the parts affected are localized within a small number of modules in the system, reducing the amount of the system the developer needs to understand.

Unfortunately, through an exploratory study we conducted, we found that developers making a change also tended to have to understand "obstacle" code that was relevant both to the desired change and to some other underlying concern in the system [1]. The code related to the other concern tended to not be modularized, but rather crosscut the system structure. As one example, one developer wanted to change the way user interface information was passed around in a distributed system. As would be expected, this change involved testing the user interface code after the change. However, in addition, it also required testing that distribution was still working after the change. The distribution code was obstacle code.

Hitting obstacle code complicates a developer's task because the developer must understand the intent of the obstacle code. If the intent—the reason for the code—is understood, a developer can make a better estimation of the depth to which the obstacle code must be followed and understood throughout the system. For anyone who has ever read code written by someone else, understanding the intent is obviously difficult! At best, a developer may find a comment close to the obstacle code explaining the intent. Unfortunately, this is rarely the case.

To help developers facing this problem, we have been working on a new concept for linking design rationale, or intent, to source code called Design Rationale Graphs (DRGs). A DRG structures information from design documentation and links the information to appropriate points in a system's codebase. Our current work on DRGs focuses on the structuring and linking of documentation in the form of Design Patterns [2] to source code.

A snippet of the DRG created from the text of the Visitor [2] Design Pattern is shown in Figure 1. Primary nodes (coloured blue) in the DRG represent design elements, such as the **`accept`** operation: Edges and other nodes represent relationships between the design elements distilled from the design documentation text, such as the **`accept`** operation taking a **`Visitor`** object as a parameter.

Given a set of DRGs for a system, we believe it is possible to use an arsenal of existing source code analysis tools to connect detailed nodes in the DRGs to the source code. The presence of such links would enable the use of DRGs to understand the intent of a piece of code. A developer

hitting obstacle code when making a change could then follow links from the affected code up to a corresponding DRG to understand why the code exists. For instance, it might be possible to determine that a particular piece of code exists to enhance the performance of a distributed system by minimizing the amount of data passed through the system.

With this approach, developers may have some hope of understanding why a concern is present in a system. Understanding the "why" is an important step when determining if and how a concern should be modularized through either reengineering or the use of an advanced separation of concerns technique, such as aspect-oriented programming [3] or hyperspaces [4]. Furthermore, by following links up to the "why" of a concern and then back down to the code, developers may also be able to elaborate all other places in the codebase related to a concern.

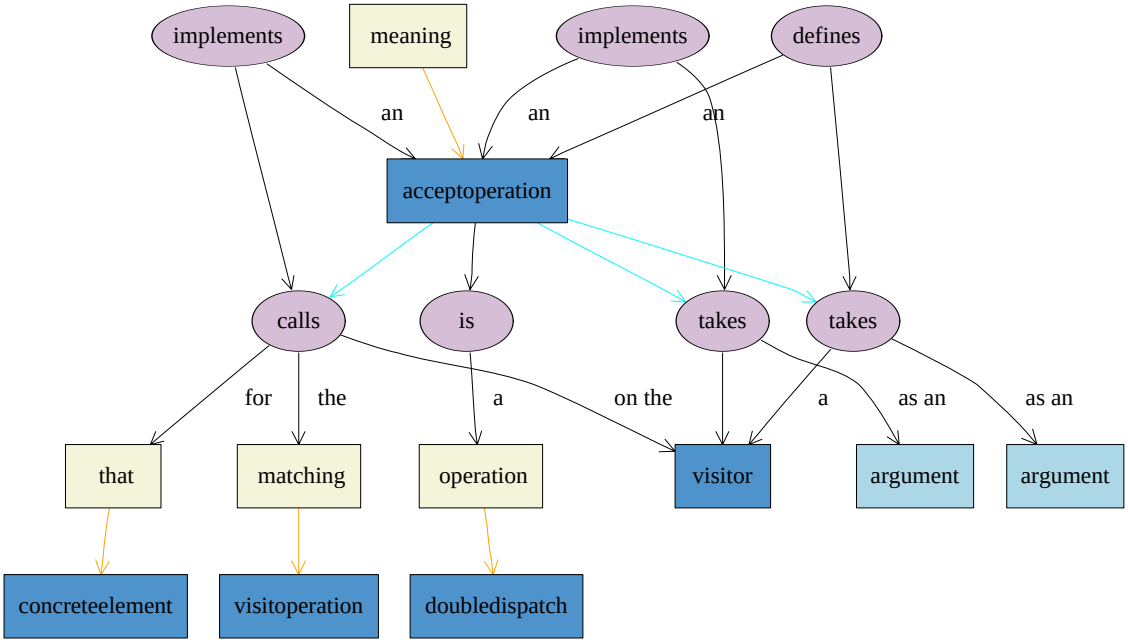

Figure 1 DRG Snippet for Visitor Pattern

References

[1] E.L.A. Baniassad, G.C. Murphy, C. Schwanninger and M. Kircher. Where are Programmers Faced with Concerns? Position paper for the Advanced Separation of Concerns in Object-oriented Systems Workshop, OOPSLA 2000.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns, Addison Wesley, 1995.

[3] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Longtier, and J. Irwin. Aspect-oriented Programming. In *Proceedings of ECOOP '97*, LNCS 1241, pp. 220-242, Springer, 1997.

[4] P. Tarr, H. Ossher, W. Harrison and S.M. Sutton Jr. N Degrees of Separation: Multi-dimensional Separation of Concerns. In *Proc.of the 1999 Int'l Conference on Software Engineering*, pages 107-119, May 1999.