Software evolution through intentional classifications*

Mens Kim and Mens Tom

January 16, 2002

Abstract

Maintaining and evolving large software systems is hard. One of the main underlying causes is that existing modularization mechanisms are inadequate to handle cross-cutting concerns. We propose *intensional software classifications* as an intuitive, lightweight and non-intrusive means of modeling cross-cutting concerns. They increase our ability to understand, modularize and browse the implementation by grouping together source-code entities that address the same concern. In addition, the model supports the declaration, verification and enforcement of relations among intentional classifications . As such, software evolution is facilitated by providing the ability to detect invalidation of important intentional relationships among concerns when the software is modified.

Problem statement

Developing large software systems is hard. It requires a whole range of labour-intensive activities such as software understanding, browsing, implementing, restructuring, testing, debugging, evolving, and so on. Once software systems reach a certain size, the modularization constructs provided by current programming languages or environments fall short. They support a limited number of modularizations of the software only. Unfortunately, as has been recognized by the aspect-oriented software development (AOSD) community [Kic97], 'system-wide' concerns often do not fit nicely into the chosen modularizations. These concerns are said to cut across the modularization. Choosing another modularization merely shifts the problem, leading to another set of concerns that cut across the modularization. The problem is that there exists a 'dominant decomposition' into which everything else needs to be fit. Perry et. al. call this problem the *tyranny of the dominant decomposition* [TOHSMS99].

AOSD tries to cope with this problem by implementing a 'base' program (addressing the dominant concern) and several aspect programs (each addressing a different cross-cutting concern) separately and then 'weaving' them all together automatically into a single executable program. Unfortunately, AOSD still suffers from some important and non-trivial problems. For example, it is difficult to debug software written in an aspect-oriented style, because it is not trivial to trace back a bug in the executed code to the corresponding aspect program(s). Another problem is that the 'aspect weaver' often depends on the actual aspects being weaved.

Suggested solution

To solve the problem of the tyranny of the dominant decomposition, we propose a more pragmatic complementary approach. We do not add extra language constructs but provide a powerful and expressive software modularization mechanism at a more abstract level in a way that is explicitly (and enforcably) linked to the implementation. Instead of describing the dominant concern and the various other concerns in separate aspect languages that are weaved afterwards (as with the AOSD approach), we stick to the idea of having one single source-code repository. Nevertheless, we allow the software to be modularized into multiple

*Extended abstract presented at the meeting of the FWO research network on Formal Foundations of Software Evolution in Brussels on January 18, 2002 (also see `http://prog.vub.ac.be/poolresearch/FFSE/meetings/meeting2002-01.html`).

user-defined *intentional classifications* that may crosscut the actual implementation structure and that may be overlapping. The idea is that each intentional classification corresponds to an important (functional or non-functional) concern that may be spread throughout the entire implementation. Such a classification is 'intentional' in the sense that it does not have to be explicit in the actual program structure: it can be seen as a kind of 'overlay' on an existing source-code repository. It is a 'classification' in the sense that it classifies the source-code entities into different conceptual modules [MM00].

In addition to defining intentional classifications , we also propose to express, verify and enforce the important relationships among intentional classifications . As such, many assumptions that otherwise remain hidden in the software implementation are made available as explicit knowledge about the software system.

Discussion

Using intentional classifications and their relationships to make software concerns explicit increases maintainability and evolvability of the software. First of all, it enhances software understanding because the intentional classifications and their relationships provide important knowledge about where and how certain concerns are implemented and how they relate with other concerns. As such, they serve as a kind of active and enforceable documentation at an abstract level that is not explicitly available in the implementation. Second, it is easier to manage the software because the important concerns are explicit in the intentional classifications , even if they are not localized in the implementation. Finally, when the software evolves, we can use the constraints imposed by the intentional classifications and their interrelationships to verify that no 'hidden assumptions' have been broken.

The model of intentional classifications is a very simple and intuitive one since it is essentially based on simple set theory only. Another important advantage of our intentional classification approach is that it is *non-intrusive*. It can be added easily on top of an existing programming language or environment, allowing the definition of intentional classifications on top of the modularization mechanisms supported by the language.

Experiment

A full version of this paper reports on a concrete feasibility study of the practical use of intentional classifications . We applied the approach to several minor and major releases of SOUL, a medium-sized object-oriented framework implemented in Smalltalk. We use a logic language at meta level to express a variety of functional and non-functional concerns in SOUL as intentional classifications . Checking, analyzing and comparing these classifications on the different releases of the studied software system helped us to manage and understand the evolutionary software development process. In many cases, invalidation of relationships between intentional classifications allowed us to identify evolution conflicts.

Related and future work

Intentional classifications are a potential candidate to serve as an underlying formalism for an architectural model that allows us to describe software architectures in such a way that we can still check conformance of the source code to architectural constraints [MWD99, Men00b, Men00a].

The proposed approach is *complementary* to AOSD research. On the one hand, AOSD techniques can be applied straightforwardly on top of intentional classifications , for example, to generate or weave code for all artifacts that belong to a certain intentional classification [TDV00]. On the other hand, intentional classifications could be a useful way of describing aspects that occur in a certain software system (for example, as a first step in re-engineering a normal program to one written in AOSD style).

Summary

Intentional classifications are a simple, intuitive, lightweight and non-intrusive model that solve an important problem that is also addressed by the AOSD community: the idea of crosscutting concerns. It increases our ability to understand and browse the implementation by grouping together source-code entities that address a similar concern and by allowing the definition, verification, and enforcement of relations among these groups of source-code entities. By providing these extra modularization capabilities, the software becomes more maintainable. Finally, it provides support for software evolution by indicating which important intentional relationships among concerns have been invalidated upon evolution of the software.

References

[Kic97] G. Kiczales. Aspect-oriented programming. In *European Conference on Object-Oriented Programming, ECOOP 1997*. Springer, 1997. Invited presentation.

[Men00a] K. Mens. *Automating Architectural Conformance Checking by means of Logic Meta Programming*. PhD thesis, Department of Computer Science, Vrije Universiteit Brussel, Belgium, October 2000.

[Men00b] K. Mens. Multiple cross-cutting architectural views. Workshop paper vub-prog-tr-00-15, Programming Technology Lab, Vrije Universiteit Brussel, Belgium, February 2000. Second Workshop on Multi-Dimensional Separation of Concerns in Software Engineering (ICSE 2000).

[MM00] K. Mens and T. Mens. Codifying high-level software abstractions as virtual classifications. Workshop position paper vub-prog-tr-00-14, Programming Technology Lab, Vrije Universiteit Brussel, Belgium, March 2000. ECOOP 2000 Workshop on Objects and Classification: a Natural Convergence.

[MWD99] K. Mens, R. Wuyts, and T. D'Hondt. Declaratively codifying software architectures using virtual software classifications. In *Proceedings of TOOLS Europe 1999*, pages 33–45. IEEE Computer Society Press, 1999. TOOLS 29 — Technology of Object-Oriented Languages and Systems, Nancy, France, June 7-10.

[TDV00] T. Tourwé and K. De Volder. Using software classifications to drive code generation. Workshop paper, Programming Technology Lab, Vrije Universiteit Brussel, Belgium, March 2000. ECOOP 2000 Workshop on Objects and Classification: a Natural Convergence.

[TOHSMS99] P. Tarr, H. Ossher, W. Harrison, and Jr. S. M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *International Conference on Software Engineering (ICSE 1999)*, 1999.