

Multiple Cross-Cutting Architectural Views

Kim Mens*

Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2, B-1050 Brussel, Belgium
E-mail: kimmens@vub.ac.be

February 21, 2000

Abstract

With this position paper we want to make a case for the relevance of the ideas of *multi-dimensional separation of concerns* at the architectural level. Traditional approaches towards software architecture seem to take for granted that a software system exhibits a single software architecture, of which the elements map more or less directly to design or implementation-level components. We claim that multiple, potentially overlapping, cross-cutting architectural views can provide a much better insight in the overall structure, organization and functionality of a software system than one single architecture which is often strongly biased to the implementation structure of the system.

Introduction

When designing a building, architects do not make one single plan that describes the overall structure of the entire building. Instead, they use many different plans that each focus on a single aspect or view of the building: front and side views, floor plans, cross-sections, foundation, drainage system, electrical wiring, central heating, and so on. Not only do these plans address different concerns,

they are also supposed to be used by different persons: clients, bricklayers, electricians, plumbers, and so on. Many of these plans are clearly cross-cutting. For example, a client's request to add an extra window (based on a side view of the building) may require parts of the electrical wiring to be reconfigured, since the wiring is often incorporated in the walls. It may even require a partial restructuring of the building, because a window is not a carrying structure. It is the architect's job to try and construct a building that optimally satisfies the different constraints imposed and concerns addressed by all these plans.

In contrast with this accepted approach in building architecture, current approaches in the domain of software architecture [8, 10] often assume that software architectures have a direct mapping of their architectural elements to source-code, design-level or physical artifacts and their dependencies. We refer to such architectures as *application architectures* because they focus on the actual implementation structure of a software application. Application architectures describe what the implementation components are and how they are interrelated.

Although such application architectures provide good insights into the structure of a software system and thus facilitate detailed design and implementation as well as evolution and maintenance of the system, there is,

*Research funded by the Brussels' Capital Region (Belgium) and Getronics Belgium.

in general, no reason why a software architecture *should* resemble the application structure. The building blocks of a software architecture are merely abstract concepts that are meaningful for the application domain. An architecture is a relation (or structure) over such concepts. Therefore, apart from the application architecture, many other kinds of architectures are imaginable and needed. For example, an architecture focusing on specific aspects of the system such as user interaction, distribution and error handling, or even architectures addressing domain-specific concerns such as rule-based interpretation (in the domain of rule-based systems). Such architectures, however, often *cross-cut* the application structure or application architecture. Furthermore, even the application structure itself can be described from different viewpoints, for example, from a data-flow or from a control-flow perspective.

The idea of having not only an application architecture but also many other overlapping and cross-cutting architectures that address specific concerns is obviously inspired by the research on *aspect-oriented programming* (AOP) [5]. AOP tries to solve the problem that when a software system is structured according to its base functionality, adding aspects which cross-cut this structure often requires system-wide changes. This problem is caused by what Tarr et. al. [11] call the *tyranny of the dominant decomposition*: typically, a software system is decomposed according to one ‘dominant’ concern and other concerns that cross-cut this basic functionality are difficult to incorporate in the software. In AOP, there is no dominant concern. The base program and several aspect programs are all implemented separately and are then merged into one single executable program. In the same spirit, Ossher and Tarr suggest to adopt a software development approach which allows a simultaneous decomposition according to multiple, potentially overlapping concerns or dimensions. Approaches such as AOP, subject-oriented programming

[3], adaptive programming [7] and composition filters [1] can, in some sense, be regarded as a special case of their approach.

With this position paper we want to illustrate the relevance and importance of the above ideas at the architectural level. In fact, we make two different claims:

1. A software system does not necessarily have one single (dominant) architecture, but should be described by several (potentially overlapping) architectural views, each providing their own perspective on the software system.
2. The elements in an architectural view do not need to map directly to implementation or design-level components but may actually cross-cut the software.

Terminology

Many authors [2, 4, 6, 8, 9] consider a software architecture merely as a structural description of the interaction among the software components of which the system is constructed. In this view, there is no objection against using the term *component* at the architectural level. However, because of our position that architectural views do not necessarily require a direct mapping of the architectural elements to design-level, implementation-level or physical components, we are *not* tempted to adopt this terminology. Not only is the term (software) *component* already heavily overloaded, most definitions of components seem to agree at least on the fact that a software component is some localized, reusable and replaceable piece of implementation of a software system. Extending the usage of the term, to denote architectural elements that cross-cut the design or implementation, would only give rise to confusion.

Instead, we prefer to use the term *concept* to denote architectural elements. This corresponds to our intuition that a software architecture expresses relations (or structure) over

abstract concepts that have some meaning for the application domain. How exactly these concepts are actually implemented is not important at this level of abstraction. So instead of talking about architectural components and connectors (as, for example, in [9]), we will talk about architectural *concepts* and *relations* respectively.

Experiments

We validate our claims by means of some experiments that have been conducted in the context of our Ph.D. research. We try to declare the architecture of some software system from different points of view, and automatically check conformance of the implementation of that system to these architectural views. The system we considered was SOUL, a rule-based programming environment, implemented entirely in Smalltalk.

Multiple architectural views

To validate our claim that a software system may have multiple, potentially overlapping, architectural views we show two complementary views for the SOUL system: the ‘user interaction’ architectural view and the ‘rule interpreter’ architectural view. Both views are valid descriptions of the system, in the sense that conformance of the system’s source code to these descriptions was verified. Due to space limitations, however, we will not discuss the details of how conformance checking was achieved.

The ‘user interaction’ architectural view, depicted in Figure 1, focuses mainly on the interaction of a user with the SOUL system. We summarize only some of its most important aspects here. The SOUL environment comes with a predefined set of *User Applications* that are activated when certain events are triggered by the user in some *Input Window*. *Auxiliary Applications* are applications that are created by *User Applications* or other *Auxiliary Applications* to do part of their

computation. After computation, a *User Application* typically produces a *Query Result*, which is not returned to the user directly, but presented in an *Output Viewer* for easy browsing and inspecting of the result.

Since SOUL is a rule-based environment, a second important architectural view is the rule interpreter architecture, depicted in Figure 2. Due to space limitations, for details on this architecture we refer to [9].

It is important to mention, though, that both architectural views are partially overlapping. For example, they both contain the concepts *Rule Interpreter* and *Knowledge Base*.

Cross-cutting architectural views

To support the claim that the concepts in an architectural view do not necessarily map directly to implementation artifacts, but may actually cross-cut the entire software implementation, we revisit the rule interpreter architectural view of the previous subsection.

When trying to map the elements of the rule interpreter architectural view to the actual SOUL implementation, we noticed that the concepts in this architectural view did not always map straightforwardly to the classes or other artifacts in the implementation. For some elements, a *cross-cutting* mapping from the architectural concepts and relations to their corresponding implementation artifacts and relationships was needed.

Consider as an example the *Rule Interpreter* concept. Intuitively, this concept corresponds to all implementation artifacts that address the concern “interpretation of queries” in the implementation of the SOUL rule-based environment. Unfortunately, these artifacts were not localized in the implementation, but were spread throughout the entire implementation. In fact, the implementation was decomposed according to the syntax of SOUL’s logic language. Every node in the abstract grammar was represented by a different class, each containing one or more methods implementing part of the interpretation

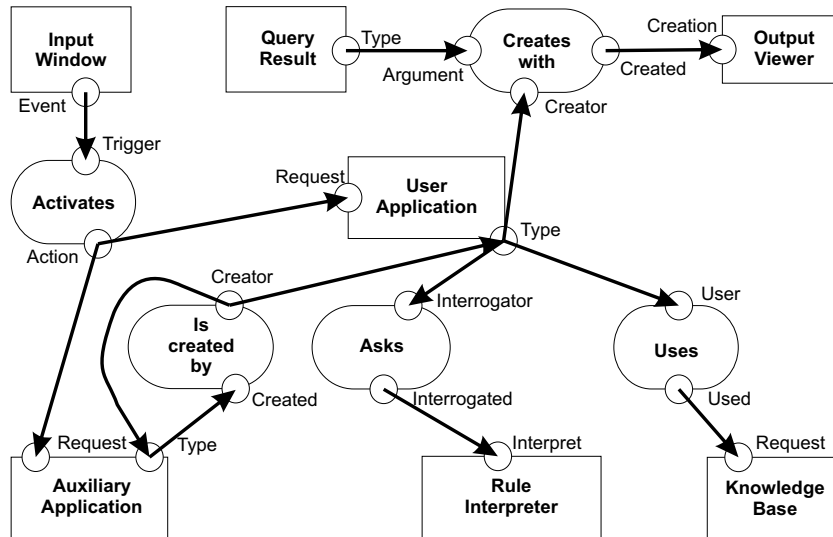


Figure 1: User interaction architectural view

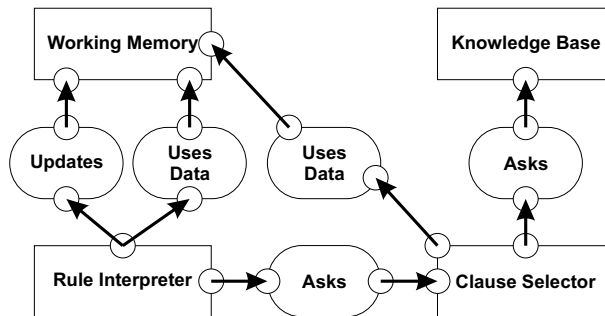


Figure 2: Rule interpreter architectural view

process. Thus, the *Rule Interpreter* concept seems to cross-cut the implementation as it is mapped to all these methods belonging to many different classes.

Conclusion

Experiments conducted in the context of our research on architectural conformance checking made us realize that an architecture which provides a high-level view of some aspect of the design of a software system, does not necessarily need to map directly to the source code, but may cross-cut it. Furthermore, many of these cross-cutting architectural views may be needed to provide a better picture of the overall structure, organiza-

tion and functionality of a software system. These architectural views may even be partially overlapping. In short, we think that the architectural research community could benefit from adopting some of the ideas of *multi-dimensional separation of concerns*.

References

- [1] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting object interactions using composition filters. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *Object-based Distributed Processing*, Lecture Notes in Computer Science, 791, pages 152–184. Springer-Verlag, 1993.

- [2] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. SEI Series in Software Engineering. Addison Wesley Longman, 1998.
- [3] W. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects). In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications, OOPSLA'93*, pages 411–428. ACM Press, 1993.
- [4] P. Inverardi, A. L. Wolf, and D. Yankelevich. Checking assumptions in component dynamics at the architectural level. In *Coordination Languages and Models, Lecture Notes in Computer Science 1282*, pages 46–63. Springer-Verlag, September 1997. Second International Conference, COORDINATION '97, Berlin, Germany.
- [5] G. Kiczales. Aspect-oriented programming. In *European Conference on Object-Oriented Programming, ECOOP'97*. Springer, 1997. Invited presentation.
- [6] J. Kramer and J. Magee. Exposing the skeleton in the coordination closet. In *Coordination Languages and Models, Lecture Notes in Computer Science 1282*, pages 18–31. Springer-Verlag, September 1997. Second International Conference, COORDINATION '97, Berlin, Germany.
- [7] K. J. Lieberherr. *Adaptive Object-Oriented Software. The Demeter Method with propagation patterns*. PWS Publishing Company, 1996.
- [8] M.-C. Pellegrini. Dynamic reconfiguration of corba-based applications. In *TOOLS 29 — Technology of Object-Oriented Languages and Systems*, pages 329–340. IEEE Computer Society Press, 1999. Nancy, France, June 7-10.
- [9] M. Shaw and D. Garlan. *Software Architecture — Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
- [10] P. Stevens and R. Pooley. *Using UML — Software Engineering with Objects and Components*. Addison Wesley, 1999. Updated edition.
- [11] P. Tarr, H. Ossher, W. Harrison, and J. S. M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *International Conference on Software Engineering (ICSE'99)*, 1999.