Micro-Architecture vs. Macro-Architecture

Joseph E. Hollingsworth
Department of Computer Science
Indiana University Southeast
New Albany, IN 47150
jholly@ius.indiana.edu

Bruce W. Weide
Department of Computer and Information Science
The Ohio State University
Columbus, OH 43210
weide@cis.ohio-state.edu

Technical Report OSU-CISRC-11/94-TR57 (November 1994)

Abstract — The field of study commonly known as "software architecture" should be split into two subareas: micro-architecture and macro-architecture. Work to date under the name "software architecture" has concentrated largely on macro-architecture. But micro-architecture, a.k.a. software component engineering, is of equal concern.

THIS PAGE INTENTIONALLY BLANK

1. Introduction

Economists discovered some time ago that one view of economics cannot be made to fit all economic situations. They noticed that different analytical techniques and models were needed when looking at economics-in-the-small as compared to economics-in-the-large [Bach 87]. This is not to say that one is more important than the other, or that one deserves more or less attention. It simply admits that the understanding and analysis of these two different classes of economic situations require different approaches in order to capture and reason about their fundamental properties.

Software engineers know that software, like hardware, has (or should have) an orderly arrangement of parts, or an architecture [Garlan 93, Prieto-Díaz 91, Tracz 93]. To date, most work on software architecture has focused on exploring a high-level view of the ways in which large systems are organized, e.g., domain-specific software architecture (DSSA).

We contend that software engineers are faced with a situation similar to that faced previously by economists: One view of software architecture cannot be made to fit all situations. Our position is that a useful approach to sort out the issues is to recognize a dichotomy in software architecture like the one in economics — to separate what we propose to call *micro-architecture* from *macro-architecture*. As with economics, this recognition does not imply that one kind is more important than the other, or that one deserves more or less attention. It simply admits that the understanding and analysis of qualitatively different software architecture situations require different approaches in order to capture and reason about their fundamental properties.

2. Where Micro-Architecture Fits In

Most work in software architecture to date is what we propose to call macro-architecture, because it deals with a high-level view of software systems and not with the details of the components in those systems. What comprises (or should comprise) work in micro-architecture? Micro-architecture is not synonymous with programming-in-the-small, i.e., with statement-level issues such as the use of structured programming techniques [de Remer 76]. Micro-architecture deals with the important issues between the high-level view of software system architecture and such low-level coding. It focuses on a finer-grain view of a system — the details of interfaces and implementations of individual components, sets of components, and how they compose and interoperate with each other.

At the macro-architecture level, a large system might be deemed to follow a general pattern, e.g., hierarchical layering of abstract data type components. There might be some interesting things that can be concluded about a system by knowing just this much — or perhaps a little more about it. But there are other interesting things about the system that can be understood only by taking a more careful view of the details, to ascertain precisely what design decisions have made it possible to construct a system with a certain macro-architecture. These design decisions characterize the component's micro-architecture.

3. Some Micro-Architecture Issues

Micro-architecture, a.k.a. software component engineering, covers many important issues, including but not limited to the following:

- What are the best ways to achieve component composability or interoperability, i.e., the composition of components with little or no glue code [Edwards 93a]?

- What are the best ways to design a component so that certifying that it possesses certain properties can be done just one time, i.e., when it is checked into a component library, instead of each time it is checked out and used in a client program [Hollingsworth 92]?

- What are the best ways to achieve plug-compatibility between multiple implementations for the same abstract component, i.e., so that a client program requires little or no modification when unplugging one implementation and plugging in another [Edwards 90]?

- What component-related programming language features are needed to best support multiple, different implementations for the same abstract component [Sitaraman 92, Dori 94]?

- What are the best ways to design a component so that client programmers have control over the component's performance, i.e., can easily tune the component with respect to space and time usage [Weide 94b]?

- Given a particular programming language (e.g., Ada, C++, Eiffel, ML) which language features should be aggressively used — and which should be avoided — when designing components to be composable, certifiable, efficient, reusable, tunable, etc.? For example, what are the best ways to use the enticing language mechanism of inheritance [de Champeaux 93, Edwards 93b]?

The notion that some language features should be used, while others should not, leads to the conclusion that even an individual software component has architectural characteristics; and that groups of components that are designed to be readily composable share a micro-architecture.

Here is a simple example of a very low-level micro-architecture issue. It is well known that one of the easiest mistakes to make during the development of code is to use a variable that has not been initialized. Many compilers try to detect this situation and issue a warning; there are also commercial tools that look for and warn against this situation (e.g., Purify). But warnings are reactive. Languages such as C++ with its constructor and destructor operations, and Ada9X with initialization and finalization operations [Ada9X 94], permit a component designer to take a pro-active approach to this problem. Therefore, part of the micro-architecture of a component (or group of components) in C++, for example, lies in the consistency of use of constructor and destructor operations.

4. Benefits from Advances in Micro-Architecture

Three benefits to studying and advancing the state of software micro-architecture quickly come to mind: new programming language constructs that support a particular micro-architecture; component design disciplines that specifically prescribe which features of a particular language to use (and how to use them), and which to avoid, when designing new components under a particular micro-architecture; and less restrictive component design guidelines that convey general approaches to designing components within micro-architectural frameworks.

4.1. New Language Constructs

The example in the previous section illustrates how language support can be provided for a micro-architectural solution to the problem of uninitialized variables. Advances in micro-architecture will lead to the need for other language-supported (or language-enforced) mechanisms that enable the design of high-quality software components. Skeptics who think the ultimate programming language already exists and that further work in this direction is pointless should carefully consider the fine points of debate in the creation of Ada9X, an ANSI standard for C++, Object Oriented COBOL, etc. Most of the suggested language changes derive directly from micro-architecture considerations.

4.2. New Component Design Disciplines

Programming languages (at least, widely-used ones) tend to evolve slowly and in response to well-understood needs of software engineers. While the community explores micro-architecture issues and develops a clearer understanding of what language support is appropriate for various approaches, there is a need for stopgap measures. Component design disciplines can be prescribed that, when followed, give rise to a specific, unique, and easily identifiable micro-architecture. For example, the RESOLVE/Ada discipline [Hollingsworth 92] includes 41 principles that very specifically guide the software engineer when designing Ada components.

4.3. New Design Guidelines

Design guidelines are less explicit than comprehensive design disciplines. When followed, though, they too can give rise to components that have easily identifiable micro-architectures. Some examples include guidelines suggesting the inclusion of component iterators [Booch 87, Weide 94a]; and the idea of recasting algorithms (e.g., sorting) as machines (e.g., a sorting machine component) [Weide 94b].

5. Conclusion

To advance the ill-defined but intuitively important field now known as software architecture, we propose that the term be defined in such a way that there are two separate but related subfields: micro-architecture and macro-architecture. We base our argument on a convenient analogy with the field of economics, and on the observation that not just large systems but even much smaller-scale software components have their own architectural features. Furthermore, when viewed from this perspective, we see opportunity for micro-architecture research to contribute to the design of future programming languages and the individual components found in common off-the-shelf libraries.

6. References

[Ada9X 94] Ada9X Mapping/Revision Team, *Ada9X Reference Manual Version 5.0*, Intermetrics, Inc., Cambridge, MA, June 1994.

[Bach 87] Bach, G.L., Flanagan, R., Howell, J., Ferdinand, L., and Lima, A., *Economics: Analysis, Decision Making, and Policy*, Prentice-Hall, Englewood Cliffs, N.J., 1987.

[Booch 87] Booch, G., *Software Components with Ada*, Benjamin/Cummings, Menlo Park, California, 1987.

[de Champeaux 93] de Champeaux, D., Lea, D., Faure, P., *Object-Oriented System Development*, Addison-Wesley, Reading, MA, 1993.

[de Remer 76] de Remer, F., and Kron, H.H., “Programming-in-the-Large vs. Programming-in-the-Small”, *IEEE Transactions on Software Engineering SE-2*, 2 (June 1976), 80-86.

[Dori 94] Dori, D., and Tatcher, E., "Selective Multiple Inheritance", *IEEE Software 11*, 3 (May 1994), 77-85.

[Edwards 90] Edwards, S.H., "An Approach for Constructing Reusable Software Components in Ada", Technical Report, Institute for Defense Analyses, Alexandria, VA, IDA Paper P-2378, 1990.

[Edwards 93a] Edwards, S.H., "Common Interface Models for Reusable Software", *International Journal of Software Engineering and Knowledge Engineering 3*, 2 (June 1993), 193-206.

[Edwards 93b] Edwards, S.H., "Inheritance: One Mechanism, Many Conflicting Uses", *Proceedings Sixth Annual Workshop on Software Reuse*, Owego, NY, November 1993.

[Garlan 93] Garlan, D., and Shaw, M., "An Introduction to Software Architecture", in *Advances in Software Engineering and Knowledge Engineering, vol. 1*, V. Ambriola and G. Tortora, eds., World Scientific, Singapore, 1993.

[Hollingsworth 92] Hollingsworth, J.E., "Software Component Design-for-Reuse: A Language Independent Discipline Applied to Ada", Ph.D. dissertation, Dept. of Computer and Information Science, Ohio State University, Columbus, OH, 1992.

[Prieto-Díaz 91] Prieto-Díaz, R., and Arango, G., *Domain Analysis and Software Systems Modeling*, IEEE Computer Society Press, 1991.

[Sitaraman 92] Sitaraman, M., "A Class of Programming Language Mechanisms to Facilitate Multiple Implementations of the Same Specification", *Proceedings 1992 International Conference on Computer Languages*, IEEE Computer Society, April 1992, 272-281.

[Tracz 93] Tracz, W., Shafer, S., Coglianese, L., "Design Records: A Way to Organize Domain Knowledge", *Proceedings Sixth Annual Workshop on Software Reuse*, IBM, Owego, NY, November 1993.

[Weide 94a] Weide, B.W., Edwards, S.H., Harms, D.E., and Lamb, D.A., "Design and Specification of Iterators Using the Swapping Paradigm", *IEEE Transactions on Software Engineering 20*, 8 (August 1994), 631-643.

[Weide 94b] Weide, B.W., Ogden, W.F., and Sitaraman, M., "Recasting Algorithms to Encourage Reuse", *IEEE Software 11*, 5 (September1994), 80-88.