Separating Fact from Fiction in Software Architecture

Nenad Medvidovic
Dept. of Information and Computer Science
University of California, Irvine
Irvine, California 92697
+1-949-824-3100
neno@ics.uci.edu

Richard N. Taylor
Dept. of Information and Computer Science
University of California, Irvine
Irvine, California 92697
+1-949-824-6429
taylor@ics.uci.edu

1. ABSTRACT

Explicit focus on architecture has shown tremendous potential to improve the current state-of-the-art in software development. Relatively quickly, software architecture research has produced credible results. However, some of this initial success has also resulted in unrealistic expectations and failure to recognize the limits of this line of research, which can result in backlash when the unrealistic expectations are not met. One solution is to attempt to clearly delineate the boundaries of applicability and effectiveness of software architectures. This paper represents a step in that direction: it dispels some common misconceptions about architectures and discusses problem areas for which architecture is well suited and those for which it is not.[1]

2. INTRODUCTION

Software architecture has become an area of intense research in the software engineering community. A number of architecture modeling notations and support tools, as well as new architectural styles, have emerged. Explicit focus on architecture has shown tremendous potential to improve the current state-of-the-art in software development and alleviate many of its problems. Architecture addresses an essential difficulty in software engineering—complexity—via abstraction and its promise of supporting reuse. At the same time, one fact should not be overlooked: software architecture is not a "silver bullet" [3], and simply introducing it into an existing development lifecycle will not fully exploit this potential for improvement.

In a few short years, software architecture research has produced credible, if not impressive, results. However, some of this initial success has also resulted in overblown claims and "hype," unrealistic expectations, and occasional failure to recognize the shortcomings and limits of this line of research. While such an attitude may be useful in gathering the initial momentum needed to put a young discipline on the map, it has the potential downside of resulting in backlash when the unrealistic expectations are not met.

We believe that, as the "first generation" of architecture research projects has become mature enough to have its actual impact evaluated, some backlash is indeed beginning to emerge. The reasons are numerous. Some projects have promised more than they have been able to deliver. Architecture is still largely an academic notion, and little resulting technology has been transitioned to industry. The focus of the research itself is often misunderstood. Architecture is sometimes incorrectly equated with architecture description languages (ADLs) and with design. As such misconceptions take hold, a danger emerges that some real benefits and positive results of architecture research will be dismissed.

It is up to software architecture researchers to alleviate this problem. We must recognize that we have failed to exploit a lot of the potential in those areas for which architecture is best suited, while promising too much, too quickly in the areas for which it is not. We must make clear that there are certain problem areas that architecture research will not be able to address and thus make explicit the boundaries of this field. This paper attempts to do so. It is partly a result of our experience with a specific project, C2 [15], and partly an extrapolation of lessons learned from an extensive study of the existing architecture work [6, 8]. While this paper may well not be the first place where some of the ideas will have appeared, they seem to have been forgotten and we believe that it is crucial to raise them again in the consciousness of architecture researchers.

The remainder of the paper is organized as follows. Section 3 addresses some common misconceptions about architectures. Sections 4 and 5, respectively, discuss the types of tasks for which architecture is well suited and those which are largely outside its scope. Discussion and conclusions round out the paper.

3. COMMON MISCONCEPTIONS

Architecture is often equated with architecture description languages (ADLs), which are only an enabler to achieve its benefits, and not the end product. Architecture is also equated with design; though related, their foci are quite different.

[1] Effort sponsored by the Defense Advanced Research Projects Agency, and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0021. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory or the U.S. Government.

ISAW3 Orlando Florida USA

Finally, existing research has focused to a large degree on architecture-based analysis. This section discusses why these are incomplete, if not incorrect, notions of architecture.

3.1 Architectures and ADLs

Software architecture is a high-level view of a system that focuses on structure, communication protocols, assignment of software components and connectors to hardware components, and so forth. Every software system has an architecture, although it may not be explicitly modeled.

An ADL, on the other hand, can be viewed as a notation that helps architects model *an* aspect of the architecture. Therefore, multiple ADLs may be used to model a single architecture. ADLs may be formal, semi-formal, or informal. Their purpose is not the same as that of programming languages: they are meant to provide a basis for early analysis of a system and for its design and implementation, but also for communication and understanding among stakeholders in a project. ADLs may be general-purpose or special-purpose modeling languages.

3.2 Architecture and Design

Current literature leaves the issue of the relationship between architecture and design ambiguous, allowing for several interpretations:

- architecture and design are the same;
- architecture is at a level of abstraction above design, so it is simply another step (artifact) in a software development process; and
- architecture is something new and is somehow different from design (but just how remains unspecified).

The second interpretation is closest to the truth. To some extent, architecture serves the same purpose as design. However, its explicit focus on system structure and interconnections distinguishes it from traditional software design, such as object-oriented (OO) design, which focuses more on modeling lower level abstractions, such as algorithms and data types. As a (high level) architecture is refined, its connectors may lose prominence by becoming distributed across the (lower level) architecture's elements, eventually resulting in the transformation of the architecture into a design.

3.3 Architecture and Analysis

A large fraction of architecture research to date has focused on the architecture as a vehicle for early system analysis. Early understanding of the properties of an application is indeed invaluable. On the other hand, there are several unresolved issues associated with architecture-level analysis. Addressing these may require a shift in the current mentality.

In order to enable formal analysis, ADLs must supply appropriate formal constructs. Performing meaningful analyses may require introduction of constructs into an ADL that are perhaps premature for the current abstraction level. This is also likely to hamper the different stakeholders' understanding of the architecture. A more balanced approach for evaluating an architecture, which couples informal "inspections" of the architecture with more powerful formal analyses may be preferable: it would result in a less formal architectural model that would focus attention on the "quality," elegance, and suitability of the architecture for the given problem.

Another critical issue is that, once a property of an architecture is established, one must also ensure that that property will be preserved in the design and implementation. Existing architecture research has mostly failed to address this problem. Better and more practical refinement techniques that would ensure consistency among the architecture, design, and implementation are needed.

4. WHAT ARCHITECTURE IS GOOD AT

In this section we discuss the tasks that arise in the process of software development for which architectures are well suited. These are the strengths of architectures and should be the primary focus of researchers and practitioners. In general, architecture provides a solid basis for *large-scale* development of *distributed* applications, using *off-the-shelf* components and connectors, implemented in *multiple programming languages*. It also has a number of additional benefits. Several are discussed below.

Architectures are a good *communication facilitator* among the different stakeholders in a project. They are at a sufficiently high level of abstraction to enable understanding by the non-technical stakeholders, such as managers or customers. Architecture arises from an analysis of the requirements and of application domain characteristics and is the earliest model of a solution to the customer needs [16]. For this reason, it is important that architectural descriptions be simple, understandable, and possibly graphical, with well understood, but not necessarily formally defined, semantics. Architectural models can also present multiple views of the same architecture [4], e.g., a high level graphical view, a lower level view with formal specifications of components and connectors, a conceptual architecture, one or more implementation architectures, the corresponding development process, the data or control flow view, and so on. Different stakeholders (e.g., architects, developers, managers, customers) may require different views of the architecture. The customer may be satisfied with a high-level, "boxes and arrows" description, the developers may want detailed component and connector models, while the managers may require a view of the development process.

One of the greatest benefits of architectures is their *separation of computation from interaction* in a system. Software connectors are elevated to a first-class status, and their benefits have been demonstrated by a number of existing approaches [1, 13, 15]. Explicit connectors remove from components the responsibility of knowing how they are interconnected and minimize component interdependencies. Changes to a system's functionality are thus limited to the components. All decisions regarding communication, mediation, and coordination in a system are isolated inside connectors [12].

Architectures are an ideal platform for supporting *coarse-grain software evolution*. This includes adding, removing, and replacing components and connectors, while ensuring structural and behavioral properties of an application.[1]

Architecture-based evolution can occur both at system specification time [9] and at its execution time [10]. Although individual components and connectors can also evolve using approaches such as subtyping or inheritance [5, 7], architecture is generally not a good foundation for fine-grain evolution that requires replacement of individual source code statements.

Architecture affords software engineers a lot of flexibility because of its *separation from implementation.* This allows an architect to experiment with various configurations (architectural plug-and-play) and enact "what if" scenarios rather inexpensively. Architectural analysis tools can then indicate potential problems early enough in the lifecycle so as to minimize the costs of correcting them. In order to fully exploit this separation, a reliable mapping from the architecture to the implementation is needed. This is related to architectural refinement, discussed Section 3.3, and remains an open area of research.

Finally, architecture is the appropriate level of abstraction at which invariant properties of a problem *domain* or rules of a compositional style (i.e., an *architectural style*) can be exploited and should be elaborated. Doing so for domain-specific architectures (DSSAs) results in a reference architecture that is reused across applications in the domain. Doing so for a style results in a set of heuristics that, if followed, will guarantee a resulting system certain desirable properties. DSSAs and styles also enable easier communication among stakeholders. For example, just saying that a system is in the "client-server" or "pipe and filter" style implies a certain structure and a set of properties without having to discuss its details. DSSAs and styles have the potential to enable code generation for canonical components and connectors, as well as the "glue" code to compose them in a system. Substantial experience in a particular domain and/or style can be used as a basis for generalizing the given approach to other problem areas [14].

5. WHAT ARCHITECTURE IS *NOT* GOOD AT

The previous section has outlined a set of benefits developers can derive from an explicit architectural focus. However, all of a project's needs may not be addressible by architecture. Architecture alone certainly cannot compensate for problems in requirements acquisition, project scheduling and budgets, organizational issues, such as poorly trained employees, inefficient process, and so forth. There are also some technical areas for which architecture is not well suited.

As already discussed, architecture cannot be effectively employed to support *fine-grain evolution.* Architecture assumes that applications are constructed from building blocks that are above the level of a source statement. If the evolution of a given system predominantly occurs at the source code statement level, then architecture is not the appropriate abstraction for supporting that evolution.

In general, architecture cannot guarantee the *properties of the implemented system.* Properties like reliability or performance cannot be fully assessed at the architectural level, although the characteristics of the architecture and/or the style can provide an indication of how the system will behave. A system's implementation can be modified independently of its architecture; many modifications (e.g., evolution at the source code level) will not have any effect on the underlying architecture, thus the architecture will not aid developers in preventing errors. If we disallow the direct modification of an implementation, we essentially reduce architecture-based software development to a variant of transformational programming [11], thus inheriting all of its problems and limitations, with the added problem of scale. Also, unless specific techniques are developed to support architecture-based evolution, and particularly runtime evolution, a system may have to be entirely regenerated every time its architecture is modified. A related issue is *source code optimization.* Architecture is not the correct abstraction level for this task; the right place to do so is code itself.

Simply employing an explicit architecture does not guarantee a development *process* that will ensure the quality of the resulting system. "Architecting" is only a step in the software engineering lifecycle. Certainly, architecture can be a guide for the process in that it contains a conceptual, high level view of the system. However, enacting an effective and efficient process to transform that architecture into a running system then becomes the responsibility of the management and developers.

A problem that is actually exacerbated by architectures is *traceability* across the different aspects (views) of a system and different levels of abstraction. As discussed above, a software architecture often consists of multiple views and may be modeled at multiple levels of abstraction (Figure 1). We call a particular view of the architecture at a given level of abstraction (i.e., a single point in the two-dimensional space of Figure 1) an "architectural cross-section." It is critical for changes in one cross-section to be correctly reflected in others. A particular architectural cross-section can be considered "dominant," so that all changes to the architecture

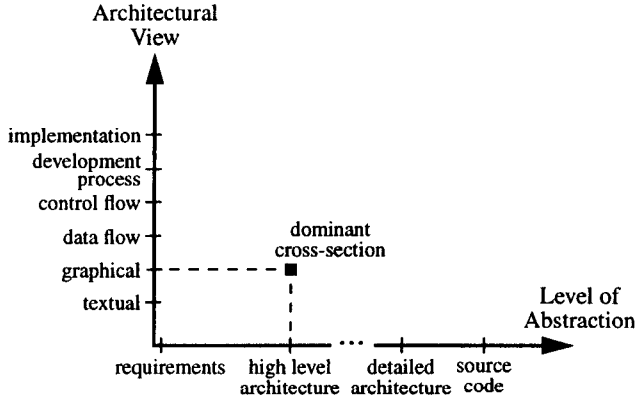

Figure 1: Two-dimensional space of architectural views and levels of abstraction. The vertical axis is a set of discrete values with a nominal ordering. The horizontal axis is a continuum with an ordinal ordering of values, where system requirements are considered to be the highest level of abstraction and source code the lowest. One possible dominant cross-section (graphical view of the high level architecture) is shown.

[1] Batory and O'Malley quite appropriately refer to this as the "Lego paradigm" in software development [2].

are made to it and then reflected in others. More frequently, changes will be made to the most appropriate or convenient cross-section. Traceability support will hence need to exist across all pertinent cross-sections.

However, the relationships among architectural views are not always well understood. For example, while it is typically straight forward to provide support for tracing changes between textual and graphical views, it may be less clear how the data flow view should affect the development process view. In other cases, changes in one view (e.g., process) should never affect another (e.g., control flow). An even bigger hurdle is providing traceability support across *both* architectural views and levels of abstraction simultaneously. Finally, although much research has been directed at methodologies for making the transition from requirements to design (e.g., OO), this process is still an art form. Further research is especially needed to understand the effects of changing requirements on architectures and vice versa.

Finally, we revisit an issue discussed earlier: *formalism*. Formalism has been extensively adopted and employed in architecture research to date, resulting in a number of benefits. However, too often this has been at the expense of understanding, communication, and cognitive support for developers. As a bridge between requirements and design, architecture is the artifact that will need to be understandable to a very diverse set of stakeholders. By removing ambiguity, formalism can aid understandability. On the other hand, some decisions will purposely be left unspecified at the architectural level; formalism typically does not allow for such incompleteness. Additionally, it is still unclear how much formalism is adequate at the level of architecture and what types of formal methods are best suited for the needs of architecture modeling.

6. CONCLUSIONS

Software architectures show great potential for reducing development costs while improving the quality of the resulting software. However, this potential cannot be fulfilled simply by explicitly focusing on architectures, just like a new programming language cannot by itself solve the problems of software engineering. A programming language is only a tool that allows (but does not force) developers to put sound software engineering principles and techniques into practice. Similarly, one can think of software architectures as tools that also must be supported with specific techniques to achieve desired properties. As is typically the case with tools, software architectures are much better suited to solving some types of problems than others. Understanding the types of problems to which architectures can be applied most effectively will help practitioners maximize the benefits of an architecture-centered approach to software engineering.

7. REFERENCES

[1] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.

[2] D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology*, October 1992.

[3] F. P. Brooks, Jr. Essence and Accidents of Software Engineering. *IEEE Computer*, April 1987.

[4] P. B. Kruchten. The 4+1 View Model of Architecture. *IEEE Software*, November 1995.

[5] N. Medvidovic, P. Oreizy, J. E. Robbins, and R. N. Taylor. Using Object-Oriented Typing to Support Architectural Design in the C2 Style. In *Proceedings of the Fourth Symposium on the Foundations of Software Engineering*, San Francisco, CA, October 1996.

[6] N. Medvidovic and D. S. Rosenblum. Domains of Concern in Software Architectures and Architecture Description Languages. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, Santa Barbara, CA, October 1997.

[7] N. Medvidovic, D. S. Rosenblum, and R. N. Taylor. A Type Theory for Software Architectures. Technical Report, UCI-ICS-98-14, Department of Information and Computer Science, University of California, Irvine, April 1998.

[8] N. Medvidovic and R. N. Taylor. A Framework for Classifying and Comparing Architecture Description Languages. In *Proceedings of the Sixth European Software Engineering Conference together with the Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Zurich, Switzerland, September 1997.

[9] N. Medvidovic, R. N. Taylor, and D. S. Rosenblum. An Architecture-Based Approach to Software Evolution. In *Proceedings of the International Workshop on the Principles of Software Evolution*, Kyoto, Japan, April 1998.

[10] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-Based Runtime Software Evolution. In *Proceedings of the 20th International Conference on Software Engineering (ICSE'98)*, Kyoto, Japan, April 1998.

[11] H. Partsch and R. Steinbruggen. Program Transformation Systems. *ACM Computing Surveys*, September 1983.

[12] D. E. Perry. Software Architecture and its Relevance to Software Engineering. *Coord'97*, September 1997.

[13] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, April 1995.

[14] R. N. Taylor. Generalization from domain experience: The superior paradigm for software architecture research? In *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, San Francisco, CA, October 1996.

[15] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, Jr., J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*, June 1996.

[16] W. Tracz. DSSA (Domain-Specific Software Architecture) Pedagogical Example. *ACM SIGSOFT Software Engineering Notes*, July 1995.