Unifying Software Architecture with its Implementation

EOIN WOODS
Artechra
AND
NICK ROZANSKI
Artechra

This paper proposes a research challenge to narrow the gap between the architecture and implementation of a software system. We discuss the situation today, where little of a system's architecture and design is directly represented in the source code of its implementation and explore why this is a problem and the benefits that might flow from narrowing this gap. We then discuss how the situation could be improved by the creation of technologies that allow the design and architecture of a system to be directly related to its source code and outline the requirements that we believe that any such technology would need to fulfill. Finally, we consider the likely results of widespread application of such a technology and the benefits that would follow.

1. INTRODUCTION

It is widely accepted today that the architecture of a software intensive system is a major determinant of the qualities that the system can deliver for its users, acquirers and other major stakeholders. It is also widely recognized that every system has an architecture, whether or not there is an architectural description that exists outside of the system's implementation [2]. However, in spite of these insights, there is usually little concrete evidence of a system's architecture in its implementation, other than sometimes a somewhat ephemeral sense of overall order and regularity in the implementation. However, while the presence of the architecture may be felt in the implementation, it can be very difficult to discern exactly what the architecture defined, let alone why.

One of the reasons for this disconnect is a technological mismatch between the design activities and the implementation activities. In practice, the architecture and design of a software system is normally represented using semi-formal notations like UML or simply ad-hoc diagrams. However the architectural constructs usually can't be directly represented in implementation languages and technologies, so it's often hard to know how closely the implementation matches the design intent. Thus, when people try to recover the architecture of a system from its code, there is little to go on and they usually have to rely on naming conventions, source code tree layout, physical deployment packaging and other imprecise and unreliable sources of architectural information.

This lack of architectural information in the implementation means that much of the architectural description of a system stays in the designer's head. Given the importance of a system's architecture this is a very undesirable state of affairs and suggests the possibility of fruitful academic research and industrial/academic collaboration.

In the remainder of this paper we discuss this problem in more detail and consider how it could be solved and the research that might help in advancing towards this goal.

2. STATE OF PRACTICE

Why is the Existing Research Insufficient?

On the whole, existing academic research in software architecture does not dwell on the problem of representing architectural design information in the implementation. Some research areas that are concerned with related topics are architectural description languages (ADLs), code generation and architectural evaluation techniques. However none of these areas have made a great impact on the problems we outline here, and we'd don't believe that this is likely to change.

A number of *architectural description languages* have been proposed [7] but most haven't been widely used in industry, particularly for information systems. We'd argue that this is generally for good reason as the ADLs often don't address the kind of descriptions that industrial software architects want to create and have little tool support.

The area of *code generation* has recently matured past the point of trying to use UML as a programming language and onto using domain specific languages (DSLs) as the basis for code generation. However there are two reasons why we don't think that code generation (as it is today) will solve the problems we describe: firstly a lot of code generation languages are at a relatively low level of abstraction and secondly, the code generation DSLs that we have encountered don't capture a lot of architectural information, suggesting that the code generation community's attention is elsewhere.

The software architecture research community has designed a number of *architectural evaluation techniques* (such as SARA [10] and ATAM [4]). Again though, while these can be valuable techniques, they don't address the problems we're talking about here; these techniques are really aimed to evaluate the quality of an architectural design before implementation and they don't fundamentally change the problem that there is no representation of the architecture in the running system.

Practicing software developers, including those designing systems – software architects – tend to think primarily in terms of concrete technical structures (libraries, modules, queues and topics, servers, and so on). This isn't really very different to how (for example) electronic engineers think about their work too, but a common disconnect between practice and research is that software engineering researchers tend to think in much more abstract terms such as components, connectors and even decisions.

That said, many well-designed industrial software systems do have an abstract architectural structure and concepts such as components, layers and interactions are commonplace in industrial software architecture descriptions. Some of the widely read patterns texts (e.g. [5] and [3]) illustrate common kinds of architectural structure, most of which are concerned with organizing the system "components" into a coherent structure.

What Happens in Practice?

In many of the projects we have seen there is often an unhelpful division between the architecture and design work and the implementation of the system. Even when the same people are involved, it is often the case that most of the design thinking happens at the start of the work cycle (iteration, delivery tranche) and then when the design is implemented, the focus changes to low level concerns and there is a real danger that the implementation will drift from the architecture. We think this is partly because it is difficult to visualize the architecture in implementation terms when implementing it.

When people do record their design in a formalized way, this will typically be done using "boxes and lines" diagrams in a drawing tool or using the ubiquitous UML notation. More common still is for the design to be captured in a transitory form such as diagrams on a whiteboard [1] or a Wiki, which is unlikely to be maintained over time. In either case though, the design has no direct representation in the code, so when you look at the code, you have to perform a lot of mental mapping to and from the design ideas to the implementation artifacts (or "guesswork" as it's more commonly known).

As far as we are aware, as practitioners, there isn't a standard widely used approach for solving this problem in industry. In practice, software architects use a number of techniques for capturing some of their design information in the implementation, including:

- *Naming* – using naming conventions and structures on software entities such as packages and namespaces, classes and modules, in order to show the existence of structural entities such as layers, patterns and component types;
- *Simple Component Models* – the use of a number of component models (such as Spring and EJB in the Java world) have become commonplace and architects often use these structuring mechanisms to represent their architectural structure.
- *Metadata* – it has become possible to include meta-data in some implementation languages (.NET attributes, Java annotations) and these are occasionally used to "tag" elements with architectural information.
- *Comments* – comments can be used in a limited way to represent design information, particularly the use of structured comments (like Javadoc and Doxygen).
- *AOP* – aspect oriented programming technologies (e.g. [6]) offer new possibilities for isolating crosscutting mechanisms in their own components.

While all of these techniques can be useful in certain situations, they all have significant practical limitations and there are no widespread conventions for using them. This means that the architect is left to invent their own approach every time, which then has to be understood afresh by those who are to use it, and isn't usable with standard tooling.

3. MOTIVATION FOR CHANGE

Many researchers have noted the benefits to be gained from integrating requirements and design, Nuseibeh's "twin peaks" model being a prominent example of this [8]. However, it isn't just requirements and architecture that benefit from being integrated this "intertwining" of the development lifecycle stages should apply to all of the development lifecycle and architecture, design and implementation should all be directly related and influence each other (as illustrated in Figure 1, taken from [9]).

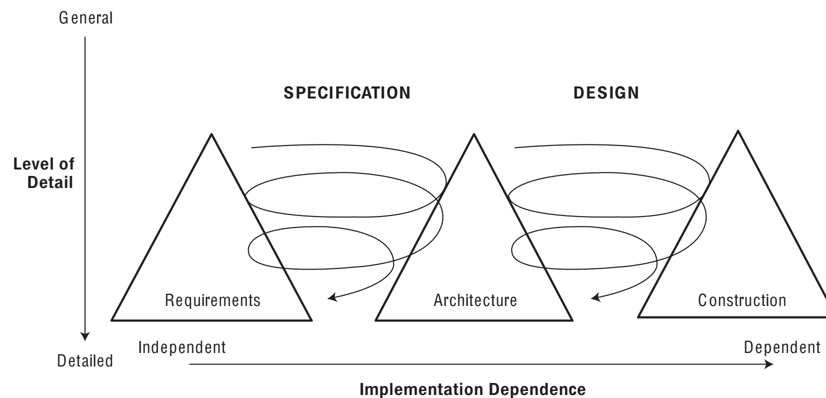

Figure 1 - Extending Twin Peaks (adapted from Nuseibeh 01])

We would like to achieve this integration between architecture and implementation because:

- It is easier to keep implementation aligned with the architecture if the processes are intertwined and the design can be seen directly in the existing implementation;
- Keeping the design visible in the code allows the design to be recovered, understood and safely extended later;
- The drift between documentation and implementation is reduced because there is enough design in the code to be clear what the design is;
- The code can be processed by machine, allowing design analysis and recovery tools.

An important part of this integration is allowing direct reference to be made between the artifacts that each activity produces, which in the case of architecture and implementation means the architecture needs to reflect the real structure of the system to be built (which we assume is already the case) while the implementation needs to be directly traceable back to the architecture (which normally isn't possible, as we've already discussed).

4. TOWARDS SOURCE CODE DESIGN REPRESENTATION

Let us consider what the key requirements of a design representation technology would be. In particular what would the requirements be for a mechanism that would be likely to be adopted in practice? We would suggest that the following key requirements are likely to be required of any widely adopted mechanism or approach:

- *Accessible embeddable notation* – some or all of the design information needs to be directly embeddable in the source code, without need for special tools. This implies a text based representation that ideally is implemented using existing extension mechanisms (e.g. Java annotations). The notation itself needs to be simple and comprehensible enough to be easily usable by casual users without special training in software architecture theory or formal notations.
- *Multi-Technology* – it should be possible to apply the technology to a range of implementation technologies, to allow for the way that most systems are built.
- *Extensible* – it should be possible to extend any notation to allow local styles and patterns to be represented (e.g. adding new component types).

- *Focus on the executable core* – the focus of the approach should probably be on the compiled and interpreted languages that the functional core of the system is implemented in, ideally with the ability to define external dependencies (e.g. data stores, message connections, RPC links).
- *Describe what Designers Design* – while hopefully extensible and quite possibly having a rather abstract underlying core the standard vocabulary provided by the technology needs to be able to describe what industrial software designers actually want to talk about, not a set of abstract research-oriented concepts.
- *Modular and layered* – in order to deal with large systems (where it will be of most benefit) the approach will need to allow for modularization and layering, allowing decomposition and composition without a lot of complexity (perhaps representation in the code modules and then separate "code" explaining how things fit together).[1]
- *Machine processable* – part of the benefit of embedding design information in the implementation is being able to process it via automated tools. The mechanism should allow the design information to be accessed and processed by automated tools at design time and ideally at runtime.
- *Availability* – the technology needs to be freely available, ideally open source, and un-encumbered by patents or unattractive commercial licensing terms.

On a practical note there are many technologies that might be useful building blocks for such an technology including JVM annotations and .NET attributes, AOP systems, structured comment systems like Doxygen, code structuring devices such as Java packages and C++ namespaces, structured data formats like JSON or XML and dynamic programming languages (like Ruby) that ease the creation of DSLs.

5. THE POTENTIAL BENEFITS

Assuming that some combination of academic and industrial research resulted in a widely adopted design representation technology, what might the benefits be? Some of the possible benefits that excite us include the following:

- *Design Recovery, Analysis and Transformation Tools* – just as the JVM and its binary formats spawned a rich collection of analysis tools, a widely used machine readable design notation would almost certainly trigger the creation of many useful design analysis and recovery tools. The information might also lend itself to more powerful system administration and operations tools.
- *Accurate Design Information* – one of the most obvious and immediate benefits would be the ability to look at a system's implementation and understand much more about its design than is possible today.
- *Diagnostic and Impact Analysis Tools* – with an accurate representation of the system's design available, it should be possible to produce more accurate diagnostic and impact analysis tools.
- *Common Language* – designers and implementers would have a common language above the level of the implementation technologies available today. Rather than having to talk about packages and Spring beans in order to be unambiguous, they could talk in terms of layers, components and connectors and still be precise.

[1] This may be an area where previous research in ADLs could be reused

- *Domain Specific Specialisations* – an extensible, widely user technology would probably spawn specialized subversions aimed at certain business domains or architectural styles (e.g. financial systems or event driven systems).
- *Criticism and the Creation of Version 2* – if the technology was really successful then it will immediately be the subject of a range of criticism as people use it, but this would then lead a new version being created to reflect lessons learned and improve on the first version. This is the ultimate indication of success!
- *Research and Practice Cooperation* – a long time disappointment of ours has been the lack of cooperation between software architecture research and practice, with neither side understanding the other all that well. The development of a technology as we describe here could benefit from the input of both communities and help to act as an example of cooperation to inspire others.

If even some of these benefits were achieved, the overall benefit to the software industry would be considerable.

6. SUMMARY AND CONCLUSIONS

In this paper we have tried to make the case that the architecture (and design) and the implementation of a software system need to be closely and obviously related to each other so that their relationship can be immediately understood by those working with the source code, so aiding comprehension and architectural compliance.

We have suggested that one way to address this problem is the development and application of new technologies that allow design information to be embedded in source code and related to higher level design descriptions, and we have tried to show the benefits that we think would follow from the successful application of such an approach.

In conclusion, we hope that a number of groups of researcher and practitioners will be interested in working on this neglected but widespread problem.

7. REFERENCES

[1]. AMBLER S., 2010, Agile Architecture: Strategies for Scaling Agile Development, http://www.agilemodeling.com/essays/agileArchitecture.htm

[2]. BASS L., CLEMENTS P., KAZMAN R., 2003, Software Architecture in Practice, 2nd Edition, Addison Wesley Professional

[3]. BUSCHMANN F., MEUNIER R., ROHNERT H., SOMMERLAD P., STAL M., 1996, Pattern Oriented Software Architecture Volume 1: A System of Patterns, Wiley.

[4]. CLEMENTS P., KAZMAN R., KLEIN M., 2001, Evaluating Software Architectures: Methods and Case Studies, Addison Wesley Professional.

[5]. FOWLER M., 2003, Patterns of Enterprise Application Architecture, Addison Wesley Professional

[6]. LADDAD R., 2003, Aspect J in Action: Practical Aspect Oriented Programming, Manning.

[7]. MEDVIDOVIC N., TAYLOR R., 1997, A framework for classifying and comparing architecture description languages, Lecture Notes in Computer Science, Volume 1301/1997, Springer Verlag.

[8]. NUSEIBEH B., 2001, Weaving Together Requirements and Architectures, IEEE Computer, March 2001.

[9]. ROZANSKI N., WOODS E., 2005, Software Systems Architecture: Working with Stakeholders using Viewpoints and Perspectives, Addison Wesley Professional.

[10]. SARA Working Group, 2002, Software Architecture Review and Assessment (SARA) Report, http://philippe.kruchten.com/architecture/SARAv1.pdf