A Small Social History of Software Architecture

Susan Elliott Sim
Department of Informatics
University of California, Irvine
ses@ics.uci.edu

Abstract

This paper presents an analysis of software architecture as social artifact, that is, something that software developers talk about and use in their work. This analysis is historical in nature, relying on interviews with software developers with experience spanning four decades and the software engineering literature. We found that 1) only large teams have architecture; 2) architecture is more easily found in discourse than in source; and 3) architecture does not happen at a fixed time in the software lifecycle. These observations taken together suggest that software is a boundary object that developers use to explain the system to each other, thereby making it possible to work together. `

1. Introduction

Since data processing became accepted business practice in the 1960s, many companies have been profitably writing software. Yet they were able to do so without the help of many of the notations and methods that are considered standard today. Even today, not all projects are developed using these methodologies, modern software engineering tools, nor software architecture technologies. But at the same time, software developers were at ease with the idea that software has architecture and they are able to talk about a system's architecture.

This paper presents an historical analysis of software architecture as a concept that is used by professional developers. The main data sources were interviews with programmers, examination of computing technology, and software engineering literature. The result is a social history because it examines how people used the term with each other and the role of architecture on their projects. This approach consequently yields a different result from one that starts from programming languages and software tools.

Our findings are as follows. One, only large systems have an architecture. While developers always used the term "design" when talking about a system, they used the term "architecture" with large systems. Two, architecture is more easily found in discourse than in source. In other words, it is difficult to point to architecture in the implementation artifacts, but, it is easier to find the architecture in documentation and in verbal explanations. Three, creation of the architecture is not limited to a single temporal phase of the software lifecycle. Despite the presence of a design phase, the architecture is created and refined throughout the life of the software.

These observations taken together suggest that software architecture is a *boundary object*, a kind of social convention to help developers understand the system well enough to work on it [5]. As a boundary object, the software architecture is a narrative that presents an idealized description of the system. Although this finding was arrived at empirically, further support for architecture as a construct for facilitating shared understanding can be found in the software engineering literature. For instance, this use of software architecture was anticipated by Perry and Wolf in their inclusion of rationale as one of the three essential components of an architecture [4].

Section 2 describes the method used to collect and analyze the data. Section 3 presents the emergence of software architecture, with a focus on industrial software developers. Section 4 explores recurring themes in the data. Finally, we conclude with a discuss ion of software architecture as a boundary object.

2. Method

This study used a historical analysis, that is, eyewitness accounts and archival resources were used. The process is closer to journalism than hypothesis testing. It is useful to study history, not merely for the facts, but because it is the context for practices today. History also tells us about ourselves because the human condition frequently brings us back to the same dilemmas.

The method used is consistent with that of a case study [7]. The initial question was "How do software developers do design, especially without the aid of rigorously prescribed methods?" The unit of analysis was a project. Design was selected as a common, neutral starting point because it did not assume a particular method or technology, which was particularly useful for inquiries regarding the 1960s.

Proceedings of the 13th International Workshop on Program Comprehension (IWPC'05)
1092-8138/05 $ 20.00 **IEEE**

The informants were not selected randomly; rather they were selected because they were able to provide insights into the development process at a given time. The interviews were conducted in the fall of 1996. However, it was only recently that the appropriate methodological tools and theoretical stance for this investigation were found.

The data were analyzed inductively to identify larger trends across time and themes across projects. In examining the answers to these questions, a programmer was considered to have used design if some effort was made to conceptualize the program before coding began and to ensure that the program was maintainable by passing on this information. While this could be done in a formal design document, more informal records such as back of the envelope scribbles, comments in the code to communicate the design, or *post hoc* documentation, were also accepted.

2.1. Interviews

Software developers with industrial experience were interviewed for this study. Each interview lasted for approximately 70 minutes. An open-ended script was used to guide the process. The script began with questions about the informant's educational and professional background, and progressed to projects that the interviewees had worked on with an emphasis on the design or architecture in the process or product. Interviewees were also asked about various resources that they found useful in learning how to do design.

Six informants participated in this study. Except for one interviewee from the 1960s and one from the 1970s, all were still working in software.

Starting in the 1960s

- "Andy" was currently a software developer at IBM Canada Ltd. He started programming in 1965 after graduating from university.
- "Jack" worked for IBM Canada starting in 1966 until he retired in 1992. He initially worked at a service bureau doing data processing for customers and later moved into software development.

Starting in the 1970s

- "Sonja" graduated with a undergraduate degree in computer science in 1972. Since then, she has been working in information systems development.
- "Alice" worked as a programmer analyst from 1974 to 1980 in various companies such as Sears Canada. She worked primarily with developing on-line information systems.

Starting in the 1980s

- "Gary" graduated from with a co-op computer science degree in 1983. He has been working as a maintenance programmer at a number of companies since.

Starting in the 1990s

- "Scott" began working as a programmer in 1993 after graduating with a computer science degree. He has worked exclusively at small companies.

3. Background: Design Over Four Decades

In this section, we give the background to our discussion by presenting a chronological slice through the empirical data. The informants gave accounts of industrial development practices spanning four decades, starting in the 1960s to 1996. The early experience informs us of processes before named methodologies, such as Structured Systems Analysis and Design (SSAD) or Design Patterns, became popularized. When contrasted with later experience, this information traces the evolution of the industry.

3.1. Prior to 1970

The two interviewees who began working in the 1960s. Both started their careers at IBM service bureaus, though on different continents. Design was done using flow charts on paper forms with plastic templates. The same tools and symbols were used for both program design and system design. A high level decomposition of a program consisted of functional units, that is, the functionality required by the user. Although the term didn't become widespread until later, these units could be labeled as modules and they corresponded to an area of responsibility for a single programmer. This decomposition was developed primarily to make the project manageable, rather than to make the code more elegant. There was also a sense of stepwise refinement. Since Wirth's work was not published until 1971 [6], this appeared to be an improvised adaptation to a complex problem. These practices extended well into the 1970s.

While the informants in our study from this era did not talk about architecture, there were others who did use the term. Among Brooks's many anecdotes regarding the development of OS/360 in "The Mythical Man-Month," there is one that refers to a meeting with Brooks, a manager of architecture, and the manager of the control program implementation [1] (p. 47).

3.2. The 1970s

Structured programming languages, such as COBOL and PL/I, structured analysis and structured design were adopted by industry during this decade. Alice and Sonja were involved in the development of an on-line catalogue system at Sears Canada. Developers closely followed the Yourdon and Constantine SSAD methodology and they

implemented the system in PL/I. Jack and Andy also reported using more structured methodologies.

In order to deal with these larger and more complex machines, such as System 360, and the software that they could support, developers had to use more rigorous processes. During this period, some concepts began to appear that are still taught today. Parnas' work on information hiding appeared in 1972 [3]. Yourdon and Constantine were publishing and teaching their method to developers by 1974 [8]. Meyer's book on modular programming appeared in 1975 [2], as did Brooks' "The Mythical Man-Month" [1].

Informants' earliest recollections of projects involving teams of many people came from this era. Jack worked on an ordering system project from 1976 to 1982. His team consisted of approximately 50 people, 12-15 of whom were coders, with the final deliverable consisting of approximately four hundred thousand lines of code. Alice's on-line catalogue system took ten people three years to develop. The final product was estimated to be several hundreds of thousands of lines of source code. A software system that Gary started maintaining in 1983 consisted of approximately three millions lines of code and came online in 1978.

3.3. The 1980s

During the next decade, developers began using "software processes" and "design methodologies" and calling them by those names. This decade was also marked by a greater awareness of maintenance issues. By the 1980s, legacy systems were large and complex enough that maintenance became a lesser evil than re-implementation.

A technological advance that occurred during the 1980s was the arrival of the personal computer (PC). PCs were relatively affordable and accessible which resulted in an explosion in computer use during this decade and the next. Small business and home PC users were looking for user-friendly shrink-wrapped software. Prior to this, there were relatively few programmers working outside of large corporations developing information systems.

3.4. The 1990s

Object-oriented languages, such as Java and C++, object oriented analysis, and object-oriented design methods were adopted. This adoption was driven by the increasing popularity of GUIs. Rather than coding them from scratch, it became easier to use frameworks and toolkits. In the past, the software to run a major enterprise application, such as billing, was several million lines of source code. Now a single spreadsheet program on a personal computer was several million lines. Even with advances in memory management, optimizing compilers, and IDEs, programming was more complex than ever.

For informants who were currently working in software, design was an integral part of their work. Two commonly cited reasons included communicating with team members, and making the code more maintainable. Scott found that when working with fickle customers, writing and revising design documents was easier than prototyping and changing a program. Gary was able to articulate and draw the architecture of all the software systems that he maintained or developed.

No further interview data was collected beyond the 1990s. However, a perusal of both popular and academic writings yield a number of recent technologies that have affected design practices; these include Web technologies, UML, design patterns, and Extreme Programming.

4. Recurring Themes

In this section, we present a thematic analysis of the data that attempts to identify commonalities across the interviewees regarding how they used and talked about software architecture.

4.1. Large Projects

The first recurring theme was that only large projects have an architecture. The size of a project is related to both the number of people, the complexity of the process used, and the size of the end-product. It seems that a project needs to reach a significant size before developers feel that they need to describe its architecture. Another way of looking at it is an architecture is a description that abstracts away details from a system and small programs don't have details remaining to remove once design is reached.

The size of project is not strictly a chronological effect because OS/360 had an architecture and it took 5000 person/years from 1963-1966 to construct the system. However, as hardware became more powerful and user expectations rose, so projects became larger, thereby making it more common that for informants to start talking about architecture on more recent projects.

4.2. It's Not In the Source Code

Architecture was more easily found to be found in discourse than in the source code. It was difficult to point to anything in the implementation that was the architecture. However, drawings or descriptions of architecture could be found in documents, and these in turn were created for communicating concepts and principles to team members.

The interviewees used the term architecture in a variety of ways. It loosely included high-level structure, the process by which the software was developed and to a certain extent the problem space

that the software fits. Andy's current project uses an "architecture document" as basis for design discussions. It includes information on requirements, data on specifications and a wish list of features. He finds this a rather confusing, unwieldy and unsatisfying document to work with. Andy feels that this document could be split up into at least three smaller, more manageable reports, corresponding to those produced in the SSAD process.

At Consumer's Gas, developers had Software Architecture Guidelines that were taught and reinforced by a Developer Support Centre. The Software Architecture Guidelines was a two-volume document; part one contained design requirements, such as data formats, safety, and security; and part two contained coding requirements, such as variable declaration, stanza ordering, and comments. Starting in 1980, all developers followed the rules set forth in the guidelines and this resulted in a high level of code re-use. Unfortunately, this standard was abandoned in 1990, along with mainframe technology and PL/I.

4.3. When Does Architecture Happen?

Architecture was not created at a particular time or time interval. The architecture starts to take shape very early in the development process and is created on an ongoing basis. Both Andy and Scott report having weekly design meetings. At these meetings, they resolved problems that were "high level issues", those that had impact on more than one programmer at a time. It appears that the architecting of a piece of software has become as interactive as coding. Designing a system on an ongoing basis may be a response to specifications being relaxed as deadlines approach.

As delivery schedules have become tighter, the attitude of "we'll get it out first, we'll get it right later" has become more common. As one would expect, less time was spent on formal design on small projects than on larger ones. This is not to say that a design was omitted altogether but that formal documents were not written. Experienced programmers were often able to put together small programs of a thousand source lines or less using only scribbles on napkins, Post-It notes and whiteboards. Occasionally, this information would be transcribed into documentation. These smaller programs tended to have a cleaner, more consistent architecture despite the lack of formal design.

5. Architecture as a Boundary Object

The recurring themes in the previous section illustrate that the concept of software architecture that is highly fluid. It's not something that can be found in the source code of a system nor at a particular time in the development lifecycle. It is created gradually over the life of a project. It can be found in documents, meetings, and whiteboards. In other words, in artifacts or rituals that are used to provide explanations to other people. These characteristics indicate that software architecture is a boundary object.

Star [5] defined boundary objects as "….objects that are both plastic enough to adapt to the local needs and …robust enough to maintain common identity" (p. 103). The plasticity of software architecture is evident in the flexibility of their origins with respect to time, place, and process. Furthermore, software architecture is robust enough to allow a team of people to work together cooperatively to bring a complex system to fruition.

A large project needs an architecture to serve as a boundary object to pull together the development team. Individual team members work on their isolated portions independently, but still need to be able to integrate their distinct parts into a whole. An over-arching organizing principle, or narrative, is needed to make sense of it all and this narrative becomes the boundary object.

6. Acknowledgements

I am grateful to all the informants for their patience and generosity with their time. This study would not have been possible without them.

7. References

[1] Frederick P. Brooks, Jr., *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition*: Addison-Wesley, 1995.

[2] Glenford J. Meyers, *Reliable Software Through Composite Design*: Petrocelli/Charter, 1975.

[3] David L. Parnas, "On the Criteria To Be Used In Decomposing Systems Into Modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1052-1058, 1972.

[4] Dewayne E. Perry and Alexander L. Wolf, "Foundations for the Study of Software Architecture," *Software Engineering Notes*, vol. 17, no. 4, pp. 40-52, October, 1992.

[5] Susan Leigh Star, "Cooperation Without Consensus in Scientific Problem Solving: Dynamics of Closure in Open Systems," in *CSCW: Cooperation or Conflict?*, Steve Easterbrook, Ed. London: Springer-Verlag, pp. 93-106, 1993.

[6] Niklaus Wirth, "Program Development by Stepwise Refinement," *Communications of the ACM*, vol. 14, no. 4, pp. 221-227, 1971.

[7] Robert K. Yin, *Case Study Research: Design and Methods, 3/e*. Thousand Oaks, CA: Sage Publications, 2002.

[8] Ed Yourdon and Larry L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*: Prentice Hall, 1985.