The Coming of Software Architecture: A Historical View

Susan Sim
University of Toronto
simsuz@turing.utoronto.ca

Abstract

Six programmers with experience spanning four decades were interviewed about their use of architecture, or lack thereof, in developing software systems. The data collected included few surprises and a history of hardware and software. The two biggest factors in the use of design are management endorsement and technological sophistication of problems and solutions. The causes and effects of these factors are discussed.

1.0 Introduction

Various software development notations and methodologies were introduced in the 1970s and 1980s. These include Structured Systems Analysis and Design (SSAD), (Yourdon and Constantine, 1985, De Marco, 1978), Structured Programming (Meyers, 1975), and Object Oriented Programming (Cox, 1984). Although, these methodologies are considered standard today, significant software was written without their help before their introduction. How was this accomplished by developers? Furthermore, even today not all projects are developed using these methodologies. Why? What are developers using in their place? This study attempts to answer these questions by looking at the practices of programmers past and present.

2.0 Methodology

2.1 Procedure

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 subjects' educational and professional background. Questions progressed to projects that the subjects had worked on with an emphasis on the design or architecture in the process or software. The direction that these questions followed was determined by the nature of the projects. Subjects were also asked about various resources that they found useful.

2.2 Subjects

Six subjects were used in this survey, two women and four men. Two started working in software in the 1960s, two began in the 1970s, one began in the 1980s and the last in the 1990s. All subjects except one from the 1960s and one from the 1970s are still working in software.

"Andy" is 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.

"Sonja" graduated from computer science at the University of Toronto in 1972. Since then, she has been working in information systems development. With some interruptions, she has been working at Consumer's Gas since 1978.

"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.

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

"Scott" began working as a programmer in 1993. He spent three years at a small speech applications development firm and is now at an even smaller firm working on Internet and World Wide Web applications.

2.3 Shortcomings

A random sampling was not used to obtain the subjects for this study. Therefore, subjects may be more opinionated on some topics than average. There are relatively few data points so second order statistics would not be appropriate in this study. This study was designed to elucidate the point of view of programmers throughout the last four decades of software development. As a result, we may see some "Dilbertism", that is the

tendency of front line workers to be biased against management. Historical accounts were given by subjects in current terms rather than the language of the era. Computer terminology has evolved over years to become more succinct and specific. Once new terms and concepts have entered into common usage, it is difficult to eschew them. Also, the use of these terms eased communication with the interviewer who had a very modern sensibility.

3.0 Findings

Data gathered in this study were consistent with the history of computing over the last four decades and a naive analysis of the software industry. Many of the processes that subjects used were common sense applied to the situation at hand in lieu of standard methodologies. This section focuses on subjects accounts of software practices, past and present. An analysis of the findings taken as whole will be presented in the next section.

In analyzing 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 structure to other programmers, or *post hoc* documentation, were also accepted.

3.1 Four Decades of Software Developers

The subjects were able to give accounts of industrial development practices spanning four decades, starting in the 1960s to the present. The early experience informs us of processes before named methodologies, such as SSAD, Object Oriented Design Patterns, or Booch, became popularized. When contrasted with later experience, this information traces the evolution of the industry. Taken as a whole, the interviews provide insight into the concomitant factors in the use of design in software development.

3.2 Use of Design Prior to 1970

The two subjects 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, i.e. not procedures but 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, both for the programmer and the supervisor, 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, this appeared to be a natural psychological adaptation to dealing with a complex problem.

However, projects tended not to have a design component in their development cycle. More often than not, programs were the responsibility of a single person. This is due in large part to the technology in use at the time. The IBM 1401 was the standard machine for business data processing in their Service Bureaus and the IBM 7044 was the corresponding machine for scientific computation in their Data Centres. Most companies did not have their own computers, nor their own developers. As a result their information technology requirements were contracted out to a large computer company. Jack recounted tales of his days spent at the IBM Data Centre on King St. in Toronto. This location had a large storefront window that allowed passers-by to see the computers in action.

The computers didn't have operating systems, as we know them. Computers were booted by flipping a number of toggle switches in a Morse code-like manner. Programs were either loaded from punch cards or from a reel-to-reel tape. A large computer had 8K of memory, while an extremely large computer had 16K of memory. In contrast, the typical handheld programmable calculator today has 32K or even 64K of memory. Programs were written on coding forms because interactive terminals didn't exist. The

programming language for the 1401 only had two control structures: if-then and goto. A large program had 1000 lines of code. Source code was first converted to punch cards. These source punch cards were compiled into an object deck. These object decks were run by prepending some header cards to the stack and loading the whole pile into a computer.

These practices extended well into the 1970s. Subjects who worked during this era enthusiastically shared their artifacts, such as coding forms and punch cards. The prevalent computer technology, both of hardware and software, did not require standardized coding practices, particularly when program was written by a single programmer. For Alice, the bottleneck when coding was how quickly she could write. It took her more time to commit the code to paper than it did to generate it in her head.

3.3 Design in the 1970s

Structure was the dominant software advance of 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 Structure Systems Analysis and Design methodology and they implemented the system in PL/I. Jack and Andy also reported using more structured methodologies.

The adoption of these methods can be attributed in large part to the maturation of computer technology. IBM's System 360 was announced in 1964. It had 256K of memory; this was as much as thirty times more than the typical machine of the era. It also had a modern operating system that was manipulated using JCL (Job Control Language). JCL was confusing, complex and arcane. Time sharing systems appeared around 1975. These terminals allowed programmers to input code directly to the computer. While these computers were still shared resources, they were a big step up from the coding forms.

In order to deal with these larger and more complex machines (and the software that they could support), developers had to use more rigorous processes. This new technology emphasized to industry the importance of techniques and programming languages that academics had been advocating for the previous ten years. During this period of change and acceptance of structured methodologies, some concepts that are still taught today began to appear in the literature. As mentioned before, Wirth's work on stepwise refinement appeared in 1971. Parnas' work on information hiding appeared in 1972. Yourdon and Constantine were publishing and teaching their method to developers by 1974. Glenford Meyer's book on modular programming appeared in 1975.

Although large software systems existed before this decade, this is when subjects first spoke of projects involving teams of many people. A team could have as many as fifty members, although teams of approximately ten were more common. Jack worked on an ordering system project from 1976 to 1982. His team consisted of approximately 50 people, 12 to 15 of which were coders. The final deliverable had 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 maintained starting in 1983 became "live" in 1978. By the time he came to work on the system, it has approximately three million lines of source code.

3.4 Design in the 1980s

During the next decade, programming languages, analysis and design techniques changed from focusing on structure to abstraction of structures. Developers were using "software processes" and "design methodologies" and calling them by those names. This decade was also marked by a greater awareness of maintenance issues. The large software systems developed in the 1970s now had to be kept up to date and in good working order. In the past, a piece of software was more likely to be re-written than analyzed and maintained: The older software was relatively small and the source code fairly

incomprehensible so it was often easier to start over from scratch. By the 1980s, legacy systems were large and complex enough that maintenance became the lesser of two evils.

Gary was a maintenance programmer for a software system that managed term deposits, guaranteed investment certificates, and the like for major banks. There was a common core to the system plus a large component for each client. Over time as more bugs were uncovered and new banks contracted their services, the software grew to several million lines of source code. One of their customers, a major chartered bank, decided that they wanted to bring this branch of their data processing in-house. The customer spent over three million dollars trying to re-implement the system and failed. In the end, they bought the core services and their customer specific component. Gary reported that the most common bug, accounting for one third of batch job errors, could be attributed to a date anomaly, such as February 29.

A technological advance that occurred during the 1980s was the advent 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. Software was developed for a particular customer or purpose. Now, software was developed for the purpose of being sold in a shrink-wrapped box to the faceless masses. The combination of PCs and GUIs was the motivation behind the next change in software engineering seen in the 1990s.

3.5 Design in the 1990s

If “structured” was the software buzzword of the 1970s, then the corresponding one in 1990s was “object-oriented”. Object-oriented languages, such as Smalltalk and C++, object oriented analysis and object-oriented design methods were adopted. Modular programming advocated hiding procedures inside modules. Object-oriented programming takes this one step further and hides both procedures and data inside

modules. This adoption was driven by the increasing popularity of graphical user interfaces. Rather than coding windows, icons, menus, and pointers from scratch, it became easier to use overlays, templates or objects and inherit this functionality. 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 automatic memory allocation, optimizing compilers, and integrated development environments, programming was more complex than ever.

Subjects who were currently working in software seemed to understand the value and importance of design. 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 that prototyping and changing a program. Gary was able to articulate and draw the architecture of all the software systems that he maintained or developed. However, these same programmers were able to name colleagues who were quite cavalier about design, documentation or commenting their code. The subjects attributed the behaviour of these “rogue” programmers to a variety of causes which included, a lack of a good computer science education, the fact that they were consultants with no vested interest in the future of the project, or simple possessiveness.

4.0 Discussion

4.1 Architecture in Development

All developers interviewed used design as part of the software process, almost in spite of management in some cases. Only currently working programmers used the term architecture. One system that Gary had to maintain, he described it as starting out with a clean architecture that was seriously damaged by underqualified programmers over the years. When asked for more details, Gary was able to draw a rough sketch of the initial architecture and describe where the structure had been violated.

The term architecture did not have a strict definition limited to the high level structure of a piece of software. It loosely included this structure, the process by which the software was developed and to a limited extent the problem space that the software fits. This appears to be a legacy of SSAD where requirements analyses and design specifications include information on the external interfaces between the system and the outside world, i.e. other systems or users. Andy's current project uses as its basis for design discussions an "architecture document". 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.

Design is no longer limited to a single temporal phase of a project. 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. Here we are beginning to the importance of management in the emphasis of design in the software process.

4.2 Management

Management was the single most important reason cited for neglecting the design during development. In the best cases, managers, both direct and upper level, were committed to a project and the use of a particular protocol. They provided enough time for all phases of the project to be completed properly. They supported some standardized methodology in the form of tools, education and support personnel. In the worst cases, managers would instruct programmers to bypass the design phase either out of ignorance or a desire for a speedy project completion. Once management initiates a project, there is a real desire to see results, and that means code. Steve McConnell (1993) refers to this as "the WICA or WIMP syndrome: Why Isn't Sam Coding Anything? or Why Isn't Mary

Programming?" He suggests three ways for programmers to deal with this: 1) Refuse such a directive and allow relations with management and their bank balance to suffer; 2) Pretend to be coding while really working on design; and 3) Educate the boss about technical processes. This is an example of the aforementioned, "Dilbertism". In addition to management's explicit directives, there are also the implicit messages about work in general and their relationship with programmers. This may encompass attitudes about workstyles, code reviews and other means to encourage pride in quality software.

In the study, there were a handful of examples where management was highly supportive good software practices. Upper level management overseeing the Sears Canada on-line catalogue system in 1974 was committed the SSAD methodology and structured programming and they went to great lengths to support this commitment. They put all developers involved on this project on a 3 month training program which included an SSAD course taught by Ed Yourdon himself. They went overseas to Great Britain to hire managers with experience using these techniques. Finally, they gave these new managers the authority to transfer to other projects developers who were unwilling to follow the new methods.

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, 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. Since then they have been trying different programming languages, software tools, hardware, and operating systems with lackluster success.

A project that Alice worked on at a major Canadian telecommunications company in 1979 skimped on feasibility, requirements analysis and design specification stages of the

project. They also used an untried development method that paired up a programmer analyst with a user for the duration of the project. While interpersonal relations could be quite good, both halves of the pair found this working model quite frustrating. In the end, the product that was delivered was bug-riddled and was quickly replaced.

4.3 Time

Subjects often cited lack of time as a reason for omitting design, but this appeared to be a corollary to lack of management support. The project that Jack enjoyed the most in his 26 years of programming was one in which his manager took great pains to ensure that front line programmers had enough time to deliver a quality product. Unfortunately, this made him rather unpopular with upper level management and the customer. In the end, they delivered a product that met all requirements with very few bugs. Incidentally, due to the way that dates are internally represented, this system will not have to deal with the year 2000 problem.

Over the last three decades, delivery schedules have become tighter and tighter. Part of this can be attributed to the speed with which technology changes, with the worst culprits being the Internet and the World Wide Web. Over the last five years, the attitude of "we'll get it out first, we'll get it right later" has become quite common. As a result, development has become demo-driven rather than product-driven. It is extremely important to develop working prototypes for marketing to show to customers or for upper level management to show to investors as quickly as possible.

4.4 Software and Technology

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 often able to put together small programs of a thousand source lines or less using only scribbles on lunch napkins, envelopes, Post-It notes or whiteboards. Occasionally, this architecture information would be transcribed into documentation. These smaller programs tended to have a

cleaner, more consistent architecture despite the lack of formal design. Perhaps a thousand source lines of code is an upper limit of how much a programmer can hold in her or his head at once.

Sometimes it is a particular design method that is omitted rather than an entire design phase. Scott recalled an occasion when marketing asked that he use Microsoft Foundation Classes in a product inappropriately so that the product literature could include the buzzwords "developed using MFC". There is also a big gap between when a developer first hears about the theory behind a particular software technology and when she or he understanding it well enough to put it into practice. In the 1970s, when the new technology was structured programming, the questions were "What makes a good module? What should I put in it? How big should it be?" In the 1990s, when the new technology is object-oriented programming, the questions are "What makes a good object? What should I put in it? How big should it be?" *La plus ça change, la plus ça même.*

4.5 Technological Change

Subjects with more than fifteen years of experience were asked to compare software developed in the past with software that is being developed today. This question was simply intended to solicit their opinion and they were free to use whatever basis of comparison that they wished. A common consensus was that compared to the 1970s, developers today are better but that their products are not. They use more sophisticated software processes and tools. They also develop larger systems with better user interfaces. However, these systems are written poorly on platforms that change extremely quickly. Programmers are under much more intense time pressures. This rate of change makes the software very difficult to maintain. Mainframes marked a long period of relative stability of hardware. They allowed analysis and design methodologies to mature and developers to evolve an understanding of large legacy systems. With rapidly changing microprocessors, networking, operating systems and user applications, we may not see another period like this in the near future.

5.0 Conclusions

5.1 Factors in the Use of Design

The two biggest factors in the use of design in development were management endorsement and technological requirements. In the case of the former, management may have implicitly or explicitly support the use of design. The most significant way that they did this was to provide sufficient time deliver a quality product. In the case of the latter, a design method was adopted when there was a proven need for it, such as when problems became too complex to solve using an existing method. Looking at the data, it could be argued that structured programming was adopted to deal with large software systems that were ushered in by the System 360 mainframes and that object-oriented programming was adopted to deal with graphical user interfaces. In other words, a particular design method was used when the problem being solved demanded it, not because the theory sounded interesting. It was often difficult to take an idea straight out of academia and apply it in practice because good tools support or even an appropriate programming language simply were not available. Consequently, there appears to be a ten year gap between the time an idea first appeared in the academic literature and the first report of its use by subjects.

5.2 Further Inquiries

In the course of several interviews, single points arose that were worthy of further inquiry. Unfortunately, there was not sufficient time in the interviews to cover any of these topics in detail. An entire study could have been done on any one of the following areas.

- For a period time in the early 1980s, Jack was an instructor for a “modernization course” that was intended to teach introductory computer science to programmers who had started working at IBM in the 1960s. The entrance requirements, material presented and culture surrounding this course were quite unusual for IBM.
- While working on the investment management software, Gary established a “Hall of Shame”, a collection of particularly grotesque or senseless bugs. His favourite

example is a one-page long if-statement to set a flag followed by an unconditional assignment of the same flag to true.

- The content and culture of the *Software Architecture Guidelines* at Consumer's Gas, where Sonja worked is quite intriguing. How was their development initiated? What was the actual rate of code re-use?
- Alice still has much of the material from the SSAD courses that she took at Sears Canada. An item of particular interest was the student manual from Ed Yourdon's course. It would be interesting to compare this material to what is taught today in information systems courses.

In order to generalize the findings of this study, a short answer paper survey could be developed and sent to a large number of programmers. The statistics derived from the survey could be compared to the data gathered in these interviews. If a correlation could be found, the combined results would be very strong because they would be supported by two very different research techniques.

Acknowledgments

I am grateful to all the subjects for their patience with a very 1990s student and generosity with their time. This study and report would not have been possible without them.

References

B.J. Cox. Message/Object Programming: An Evolutionary Change in Programming Technology. *IEEE Software*, 1(1):50-61, 1984.

T. De Marco. *Structured Analysis and System* Specification, Prentice Hall, 1978

S. McConnell. *Code Complete: A Practical Handbook of Software Construction*, Microsoft Press, 1993.

G.J. Meyers. *Reliable Software Through Composite Design.* Van Nostrand, Reinhold, 1975.

D.L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053-1058, 1972.

E. Yourdon and L.L Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Prentice Hall, 1985

N. Wirth. Program Development by Stepwise Refinement. *Communications of the ACM*, 14(4):221-227, 1971.