Evolving Software Architecture

William Lee
wwlee1@uiuc.edu
Computer Science, University of Illinois at Urbana-Champaign

21st October 2003

Abstract

Popular notion of software architecture usually describes it as an infrastructure that should not be changed after its initial creation. In this article, this article will argue against this perspective. Although good architecture should not be changed constantly, software architecture does evolve when market, knowledge, and technology change through time. In fact, those factors make architecture evolves at different pace during different stages of development cycle. There is a great need to understand and control how architecture changes. Since it is essentially for architects to step through the architectural lifecycle intelligently and correctly, perhaps further research in this area will help them do so.

1 Introduction

There are many definitions of "software architecture". Many compare software architecture to the traditional structural architecture. The architect would draw detailed blueprints on building structure way before anybody starts digging holes in the ground.

The traditional waterfall method for software development follows this paradigm. Software architects typically gather requirements from the stakeholders, go through several iterations of review and design, come up with architectures that meet those qualities, then proceed to development. This process is referred to as ABC (Architecture Business Cycle)[1]. ABC does have a feedback loop that goes from the end of the architectural design back to the root of the process. However, once you have reached the development you can not simply go back to restructure your architecture easily.

XP, or Extreme Programming, uses an entirely different approach. XP proponents dislike having one big design upfront. According to them, big designs wastes time for requirements change constantly[3]. Based on this nature, XP introduces the philosophy YAGNI (You Ain't Gonna Need It). In other words, development teams only implement the most essential features incrementally. Thus, constant refactoring plays a very important role in XP. However, there exists a great cost for constant refactoring. Architectural refactoring does not seem to be as straightforward as what people may think, since the cost for refactoring increases closer to the end of the development cycle[4]. Since your requirements have changed, and if your architecture does not meet the needs, it needs to change also. This kind of refactoring of architecture requires significant effort.

The two development models do have one thing in common: architecture may change in reaction to changing requirements. Waterfall model works well only when requirements stay mostly static, and XP proponents assume that requirements change all the time. A general rule of thumb should be: the model of development should depend on the changing frequency of the requirements. Nevertheless, in either cases architecture may need to change when there is an unexpected and substantial requirement change.

Besides requirement changes, there are many factors that cause architecture to change. Technology and the architect's knowledge may also modify the architecture in large or small ways. I would refer to the process of changing architecture as "architectural evolution". The word "evolution" is chosen for architectural changes usually do not happen overnight. In extreme cases, its evolution may span years of even decades of development.

2 Why Does Architecture Evolve

Each iteration of the software development cycle should make the product better. In order to create a superior product for the customers, marketing demands new features that hopefully map to what the consumers need.

If there is mostly-static architecture that can handle additional needs very well, you do not need to change the architecture. There are, however, cases when architects need to evolve their architecture.

2.1 The Abruptly-Changing Market

The changing market probably is the most important factor for architectural evolution. Architecture probably changes the most between software development cycles–that is, after when the project ends and before the next project starts.

Using a personal example, Sendmail Inc., the company I previously worked for, is a company that makes a collection of email software and provide support to enterprise email infrastructure. During the period that I worked there, I had gone through several iterations for architecture evolution.

A major one happened about two years ago. Sendmail used to sell a simple and fast mail store product (POP and IMAP server) that worked very well on a single machine. As the company grew, however, marketing and management suggested that the company had spent too much resources on handling small deals. A company of Sendmail's size simply could not afford the cost in an extremely competitive market. Instead, the marketing director forced the sales to focus on larger enterprise deals. This had significant implication to the mail architecture we had built before. In particular, we needed to scale that mail store product to work on multiple, enterprise-level machines. This also meant we need to support many more operating systems that run on those hardwares. The availability, performance, and security qualities suddenly required to improve tremendously.

Although the original architecture was adequate on a single machine, changing it to support multiple servers required significant engineering effort. It also put a burden on the chief architect who needed to balance against limited programmer resources and meeting tight deadlines. In the end, the architect reacted by adapting and stretching the existing architecture to satisfy the new demands. The "adaption" includes making the existing mail store more robust, create a high-performance proxy that handles load balancing, and various small enhancements that changed the architecture significantly. Due to time constraints, the architect also deferred many enterprise software requirements such as comprehensive server management and bulk loading tools.

It is quite obvious that changing market forces are always important external forces that drive architecture evolution. However, there are some cases where architecture changes from within. That is the idea behind the "Always-Learning Architect".

2.2 The Always-Learning Architect

In many projects, where there may not be a single architect, the job for creating an architecture lies on the less-capable developers. This implies that the initial architecture developed by the less experienced would leave lots of room for improvement. This is more common in smaller company where you may have several experienced programmers but no chief architect in the group. Furthermore, even very experienced architects learned their lessons during the implementation phase. As a result, architecture can change when people improve as well.

An architect that I worked with in Sendmail actually came from the consultant background. Management promoted him as the chief architect after much of his contribution to the architecture for various products while he worked as a consultant. Although his understanding to all the products in the company was quite comprehensive, he did not produce his first proposal to change the architectural design until three months after. Essentially, he took the time to obtain sufficient understanding to all the products under his supervision. As he broadened his scope, one could see that the new proposed architecture became much more complete and coherent than before.

This raises an interesting question. How does an architect "improve" on a lacking architecture that he/she had worked on before? If we look at architectural decisions as the best decisions that was made at the time, as architects improve, how does the architecture change with the architect?

2.3 The Constantly-Advancing Technologies

As technologies change, architecture may need to change too. Although the fundamentals of computer science may change very slowly through time, many architectural decisions made today differs significantly than architectural decisions that were made twenty years ago.

One example is Sendmail's flagship product, the MTA (Mail Transfer Agent) *sendmail*. When Eric Allman wrote *delivermail*, an ancestor to *sendmail,* in 1979, the hardware and network (UUCP) that he dealt with worked very differently. In the end, *sendmail* architecture carries 20 years of baggage, even with significant effort to refactor and improve the quality of the code. Also, its architecture is resilient to changes, for any radical change would almost require the world that uses *sendmail* to change as well.

When the Sendmail team and various volunteers decided to re-architect *sendmail*, it was an extremely difficult task. After all, the architectural decisions they made came from 20 years of experience of developing *sendmail*. In fact, it was quite interesting to observe how the new architecture comes into shape after the old experience clashes with up-to-date technology. For example, they decided to go with a multi-process, multi-threaded model, utilizing the more advanced and efficient multi-threading capability on modern OS's. The new version will only work on TCP/IP. There is also significant effort in splitting up the responsibility for a MTA into various components for the sake of better security. In many ways, changes in technologies influence deeply how architects think about architecture.

The Sendmail problem indicates how sometimes an architecture needs to change radically. In other times, however, with limited resources, architects need to come up with different means to evolve architecture.

3 Evolving Architecture through Patterns

Martin Fowler defines architecture as two elements: 1) the highest-level breakdown of a system into its parts and 2) decisions that are hard to change[2]. In many ways, however, people still need to make those tough decisions when architecture needs to change. Thus, we need to find a relatively smooth method do so.

We can actually see many examples in [2] when Fowler discusses tradeoffs for various architectural patterns. In fact, Fowler talks about architectural patterns in collections that rank from the most straightforward to the most flexible. Examples in [2] includes progressing from Transaction Script to Domain Model and from Page Controller to Application Controller. In some ways, this aligns nicely with architectural evolution. We can look at this with a hypothetical example.

3.1 Anna and the Architecture

Anna serves as the chief architect at a company called ToyWare. In the beginning, Anna's team needs to create a small web-portal for toy distributors to browse their toy catalog. Since ToyWare is a small company that sells only a small number of toys, Anna decides that she only needs to create a very simple site using Java Servlet with the Page Controller pattern. She uses the Transaction Script to support her backend [2] since the portal only needs simple database access. The database calls are mixed up with business logics, which are organized in a straightforward Service layers.

Then, marketing comes up with thirty new toys in various categories to sell to the distributors. When Anna evaluates the existing code, she decides that Transaction Script is not going cut it. There is plenty of duplicated DB access code already. Luckily, she can learn. While she is researching for the project, she comes upon Fowler's book. She learns about the Domain Model, Data Mapper, and Identity Map patterns and thinks they are great ways to separate their database logic from their business logic. She tells the team what she has learned, then proceed to start transforming the Transaction Script architecture into Domain Model.

Since there is not much code around, Anna's team finishes evolving their architecture to use Domain Model after a few weeks. They kept around their original service layer, since they do not want to change the external APIs. The marketing forks are happy, for now.

Unfortunately, the market environment has drastically changed after eight months. Anna's company is now expanding their product line to sell thousands of items on their web site directly to consumers. Anna also needs to comply with the new availability, security, and performance requirements. Although Anna and her team has spent much time refining their Domain Model, they still need to do a lot more.

3.2 The Tough Choice Ahead

Anna has a tough call to make. How can she evolve her architecture to meet the needs? After all, there is much code written during the last release, and there is no time she can spend restructuring the architecture. As a result, she now needs to evolve her architecture in a less aggressive manner. She learns about the capability of J2EE, and it seems like J2EE provides what she needs to meet the new quality attributes. However, there are plenty of architectural mismatches between J2EE and her original architecture.

Anna overcomes this challenge by writing various adaptors and mediators to plug her original domain model into the J2EE framework. It is not an easy tasks for it takes her and her team couple of months to do so. However, she meets all her requirements and again satisfies the requirements. Meanwhile, she keeps on evolving the architecture slowly, since it is now almost impossible to make drastic changes at this point. Nevertheless, since Anna's architecture meets most of the new requirements, only small incremental modifications are sufficient to sustain ToyWare's needs.

3.3 Completing the Lifecycle

Anna's architecture, although good enough for its time, will eventually end its life when changes in the requirements, architect's knowledge, and technology render her architecture obsolete in the future. At that point, Anna's architecture will complete her lifecycle.

From Anna's example, we learn that architecture solutions do not come into existence when the architects start their new projects. Along the way, architects need to make extremely delicate decisions as to how to best evolve the architecture in order to maximize its benefits to the company or organization. This is not an easy task.

4 Conclusion

Only architects with real vision and experience can change and evolve an architecture successfully. Although it is important to create a correct architecture initially, it is also essential that architects make the right decisions after the architecture enters its lifecycle. Intelligent architects also adjust their strategies on different stages of development. After all, between successful architectural evolution and complete project failure, there is only a fine line to walk. Perhaps further research on this topic can help align their balance.

References

[1] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. 2003.

[2] Martin Fowler. *Patterns of Enterprise Application Aarchitecture*. 2003.

[3] Ron Jeffries. What is Extreme Programming? 2001.

[4] Matt Stephens. The Case Against Extreme Programming. 2003.