Why Can't They Create Architecture Models Like "Developer X"? An Experience Report

George Fairbanks

School of Computer Science
Carnegie Mellon University
5000 Forbes Ave, Pittsburgh PA 15213 USA
george.fairbanks@cs.cmu.edu

Abstract

A large financial company, struggling with legacy systems that did not interoperate, performed a pilot project to teach software architecture to an enthusiastic application development team. Experienced mentors, including the author, worked with the application team for seven months to complete their engineering goal successfully. However, the mentors were unsuccessful in their attempt to train any of the six members of the application team to create architecture models on their own, though they were able to create them collaboratively with the mentors. This surprising result is due to the application team's strong preference for concrete artifacts over abstract ones. Even more surprising, an application developer from a different project, "Developer X", read the architecture modeling documentation on an internal website and, without mentoring, created architecture models within a few days. In light of this failure to teach software architecture, two short-term strategies are suggested for the use of software architecture in companies.

1. Introduction

Many companies struggle with systems that do not interoperate, so-called "stovepipe systems". The techniques of software architecture promise to help by providing high-level views of the system and by identifying components [10]. A seemingly reasonable approach is to train each software development team to use architecture models, provide the models to senior architects, and re-engineer applications to interoperate and share components. In contrast to earlier reports in academic environments [7, 1] the developers in our pilot project found it hard or impossible to create these architecture models without assistance. This result is important for two reasons: First, it is surprising to find that all six developers on the project had trouble learning to create architecture models. Second, the difficulty in training the necessary developers will cause organizational difficulties for companies seeking to use software architecture techniques.

I learned these lessons inside a large financial company during a pilot project to institute a software process that included architecture modeling techniques. For seven months, the author and other mentors experienced in teaching architecture and component-based development guided an enthusiastic team through their first use of the software process. The engineering goal was to specify an existing single-tier system for third-party re-implementation as a multi-tier J2EE [11] application. The educational goal was to provide sufficient training on the new software process so that the team could continue to create architecture models after the mentors had gone. The team was composed of both programmers and analysts. All knew the domain well and many were involved in the creation of the original application. This was their first exposure to software architecture and their first time creating specifications for a third-party contractor. Prior to the pilot project, the team members had proven themselves effective in designing and implementing large upgrades to the system.

The engineering goal of the project was achieved in that specifications, including architecture models, were delivered for implementation. The training goal was not achieved because when the application team worked alone they produced poor architecture models despite months of mentoring. It had been assumed that each application team could provide their architecture models to a central architecture group that would use them to redesign applications for integration. The results of this pilot suggest that this approach may not work as intended.

While the pilot project was in progress, a developer from another project, "Developer X", sent the architecture group an email requesting an evaluation of his architecture model.

His architecture model was quite good. This was of interest to the architecture group because Developer X had never received training or mentoring and had only read the architecture group's documents on the corporate intranet a few days earlier. Developer X's success suggests that the process and documentation was adequate and the failure to teach the application team to create architecture models was related to their aptitude in working with abstractions.

The remainder of this report proceeds as follows. Section 2 describes the rationale for the pilot project and new software process. Section 3 provides details on the project schedule, mentoring, and activities. Section 4 describes the observations on the project successes and failures, including the surprising success of Developer X. Section 5 investigates several hypotheses that explain the observations. Section 6 provides conclusions including two short-term workarounds and suggests long-term strategies to make software architecture accessible to more developers.

2. The Software Process

The new software process used in this project had four sources of inspiration. The first was the Catalysis [4] approach to system description, which describes how to use UML [5] precisely and relate models at different levels of abstraction. It gives guidance on how to specify and use components.

The second source of inspiration was Cheesman and Daniels' [2] work that provides advice on which levels of abstraction to choose, pragmatic UML notations, and nomenclature for creating component systems.

Third, the Model Driven Architecture (MDA) approach from the Object Management Group [9] provides ideas on how to integrate applications based upon consistent domain models across many applications.

Fourth and finally, the standard concepts from the software architecture community provided the building blocks for the construction of the architecture [8].

The new software process allows precise modeling of the business process, software architecture, components, and connectors. The notation used in architecture modeling was proprietary, but it embodied standard architecture modeling ideas like components, connectors, roles, and ports that are found in architecture description languages such as Acme [6]. By combining selected software engineering technologies, this process is designed to address the challenges that are found when integrating legacy systems. Although currently proprietary, the process is expected to be published in the future.

Product Line Architecture (PLA) techniques were not used on this project because they cannot be used to integrate existing systems, even though they can be quite effective when designing and implementing new systems [3].

3. The Project

This section describes the context, calendar, and activities during the pilot project. Due to confidentiality concerns, some details have been omitted or generalized but sufficient detail is provided to illuminate the principles involved.

Two peer groups inside the company were involved in the project. The architecture group, of which the author was a part, consisted of mentor/educators experienced in component-based development and architecture modeling. The software application team consisted of both analysts and programmers. The application team had recently decided to try outsourcing to a third party for implementation and had requested assistance from the architecture group.

The application team had been updating and maintaining a medium-sized two-tier application used by customer service representatives. With new requests to make some application functions available over the internet, the team chose to re-engineer the application to run under J2EE on multiple tiers. The engineering goal of the team was to produce architecture and component specifications to deliver to the third party implementers.

With agreement from the application team, the architecture group assigned a mentor to co-locate with the group and guide them through a software process that included the creation of architecture models of the existing system and planned improvements. The architecture group hoped that these architecture models could be used to identify reusable components and provide insight into the structure of the application to enable interoperation with other applications. Another goal was to train the application team sufficiently so that in the future they could create architecture models without the mentors present.

The sequence of events was roughly as follows. We modeled the business context and desired quality objectives. The high level architecture of the new system was designed and component choices made. Design proceeded until it was discovered that the existing system contained functionality not present in the new system design. In a painstaking process, the application team analyzed the code and transformed it into prose readable by subject matter experts. Since a rules engine was part of the new design, they converted some procedural logic into declarative rules. They created specifications for each component at two levels: a logical level and an API-level. The third party implementers then coded the specifications in Java.

When the application team was asked to produce architecture models without assistance from the mentors, the resulting architecture models lacked abstraction and made inappropriate references to concrete artifacts.

4. Successes, Failures, and Developer X

This section discusses the project successes and areas of difficulty. Generally, when the application team and the mentors collaborated they produced quality results. Without the mentors, the application team tended to produce models lacking abstraction. However, after reading the architecture group's online documentation another developer in the company produced abstract models without mentor assistance.

4.1. Success at Collaboration

The project was successful when the mentors and application team worked together to create architecture models. Frequently the mentor would facilitate a working session. Ideas would be jointly sketched, debated, reworked, agreed to, and finally recorded. In this style of interaction, the application team would supply facts about how the system worked and the mentor would incorporate them into the architecture model on the board using correct notation. This style of interaction worked because the mentors knew the software process and modeling techniques while the application team knew the problem domain of the system.

The collaboration between the architecture group and the application team was successful at producing a model of the system that neither group could have produced individually and demonstrates that the pairing of individuals with these skill sets is viable.

4.2. Failure in Abstraction

Though effective, these joint mentor and application team working sessions were not without friction. The mentor frequently acted as the gatekeeper between the architecture model and what he perceived as unnecessary details. For example, the application team preferred to show the AS/400 computer and the Sybase database as large deployed components, while the mentors instead preferred to show a batch connection to another system.

The application team trusted the mentors to judge the appropriate detail, but a lack of understanding combined with that trust may have caused problems elsewhere. The mentors discovered late in the project that the application team had not volunteered details on some of the system functions related to data caching coherency that were relevant to the architecture. This was certainly not malicious and probably based on an incorrect guess at what details were important at the architecture level.

When specifying components without mentor assistance, the application team repeatedly produced detailed specifications of a component's internal implementation but presented these as specifications of the interface. This is a problem because another programmer using the component should only want to know how to use the component, i.e., the interface specification, not how to re-implement the component. The application team did not omit the interface specification but instead delivered a duplicate of the implementation specification. After working collaboratively with the mentor, they were able to build an appropriate interface specification and agreed that the new interface specification was better.

The application team appeared more comfortable discussing their system using concrete terms. They resisted the use of abstract model attributes. In one case, data interchange requirements necessitated a single character database field to encode a combination of three independent properties. The application team resisted the abstract representation of this as three separate properties. Similarly, another database field name was "employeeFlag", but it did not identify employees nor was its datatype Boolean. The application team resisted renaming this attribute in the abstract models. At other times, the application team would convey the domain requirements in terms of database entries, as in "Q is allowed to do R if there is a matching entry in the S table," instead of in domain language "Q can do R if Q has a registered entitlement."

4.3. Surprising Success of Developer X

At one point during the pilot, a developer who worked on a separate project, Developer X, sent an email to the architecture group. Developer X had read the architecture group's internal website and had produced architectural models of his system. He asked the architecture group to review his models and, in the architecture group's opinion, his architecture models were quite good. He obtained these results without mentoring and without months of effort.

Clearly there was an enormous difference in the success of Developer X compared to the application team. The application team was given months of daily guidance from experienced mentors and feedback on how to improve their modeling yet their architecture models continued to be of poor quality. Developer X, either from former related experience or through natural ability, possessed great skill in extracting the essence of the software and creating architecture models to represent it. This idea and others are analyzed in the next section.

5. Hypotheses

Many hypotheses for the observed failures have been suggested. Many of these are listed here along with a discussion on how well the hypothesis fits with the observed facts. When necessary, additional details that support, disprove, or otherwise affect a hypothesis are also noted. The first hypotheses to be presented are the ones that have strong evidence to suggest that they are not true.

- *The application team was resistant or otherwise not trying hard enough.* From beginning to end, the team kept trying to learn the techniques and was excited to learn. This pilot project was not a case where a software process was forced upon an unappreciative team. In fact, at the end of the pilot the application team decided to keep using the new software process.

- *The team was incompetent.* When working on the new process, the application team was unsure of themselves and progressed slowly. I observed a distinct difference in the team's apparent competence when working on their usual duties. When responding to maintenance requests they were confident and quick. This team had successfully maintained the system for years and had designed and implemented significant upgrades.

- *Architecture modeling is too hard for anyone or is too hard on real-world projects.* However, when the application team and the mentors collaborated they produced good architecture models.

- *The new software process was poor, the materials were defective, or the mentoring process was flawed.* A few of the mentors believed this hypothesis for some time despite the substantial history the mentors had in teaching this material. While improvements to the process, materials, and mentoring are always possible, the mentors stopped believing the hypothesis to be a large influence once they saw Developer X's work.

- *The architecture group engaged in a game of "bring me a rock" by changing the acceptance criteria of the application team's models.* The game is impossible to win because the answer can always come back "No, a darker one" or "No, a flatter one." This is inconsistent with what the architecture group observed. The application team consistently brought the same rock, architecture models that were always at the API or data representation level.

The following hypotheses cannot be refuted given the scope of this project, but are unlikely to be true given the experience on the project.

- *The lack of software architecture ability on the pilot project was statistically abnormal and other projects will not be so unlucky.* This is indeed possible, but a cautious reader should treat this report like a sighting of a natural disaster and be aware that they can happen. This project showed six developers, not just one, who were ineffective in creating software architectures. A larger study of the developer population could better test this hypothesis.

- *"Old habits die hard" and that the team needed longer to learn the new techniques.* Perhaps a year-long project would show different results. However, if the training process takes more than seven months of continuous mentoring then it is likely not cost-effective to perform the training at all.

Based on the cumulative evidence and especially the differing abilities of Developer X and the application team, the hypothesis most likely to be true is the following.

- *Many developers are unskilled in working with abstractions, a skill necessary to create architecture models.* While I believe this based upon this pilot project and other observations, the evidence from one pilot program cannot prove this. A more conservative hypothesis that is directly supported by this pilot program is that some developers have great difficulty creating architecture models. Both hypotheses are significant enough to interfere with company-wide use of architecture models, as is described in the next section.

6. Conclusions

Viewed very generally, the finding of this report is not surprising: The project revealed a difference in ability of developers to perform a task, namely creating architecture models. However, this finding is interesting for two reasons. First, it had been assumed by many, including the architecture group, that software architecture would be a readily teachable skill. This project uncovered six competent, experienced developers who showed little aptitude for creating architectural models.

Second, if this finding generalizes then it limits the way companies can use software architects. The traditional approach to adopting a new technique is simply to train people to use it, but this pilot shows that a software architecture training program may be ineffective.

This project gave us no direct guidance on how to predict if a developer will be good at creating architecture models. As long as this is not known, companies will spend time and money attempting to train developers with poor abstraction skills. I speculate, however, that developers with poor abstraction skills self-select certain job types that allow them to emphasize their skills at manipulation of concrete artifacts. It follows that certain jobs, perhaps maintenance programming, may be disproportionately lacking developers trainable in software architecture.

Since the architecture group still believes in the benefits that software architecture can bring to the field of software engineering, we are determined to work through or around this difficulty. Turning from the problem of how to obtain good software architects, to the problem of how to use the ones that are available, I identify two short-term strategies.

6.1. Central Architecture Group

The first strategy is send architects from a central architecture group to collaboratively create architecture models with the application teams. The architects know the domain of software architecture and the team representatives know the domain of their application. As this pilot showed, they can work together for a short time to create a reasonable architecture model. This approach may not work if the ratio of architects to teams is low because the architects would be stretched too thin. Furthermore, changes to the application may warrant another collaboration session with an architect, consuming more time and effort.

As this project demonstrated, there are times when the collaboration approach is deficient. Details concerning the caching behavior of the application were not shared with the mentor because the application team did not think they were relevant to the architecture model. Also, though this project was lucky to have mentors who were experienced at facilitating modeling sessions, all projects will not be so lucky.

6.2. An Architect for Each Application

A second strategy is to require application teams to develop their own architecture models. This requires each team to have at least one software architect. While the existence of architecture modeling skills cannot be taken for granted, companies can deliberately seed each team with trained architects. This application architect has the benefit of both knowing the domain of the application as well as having architecture modeling skills, which avoids the "unmentioned details" problem that a collaborative effort can have. Also, incremental modifications to the architecture models can be done painlessly as the application evolves since an application team member can do the incremental modifications.

This approach is not without problems. It assumes that a large pool of developers can be identified and trained in software architecture techniques. At least one architect must be present on each application team, which will invariably require inter-team transfers of personnel. As mentioned above, the application teams lacking architecture skills may not be uniformly distributed across the company, further complicating inter-team transfers. Finally, the architecture models created by part-time architects may not be quite as good as models created by full-time architects.

6.3. Future Work

In the longer term, the goal should be to expand the pool of developers who can be taught to use software architecture effectively. Similar to the way that the lesson plans for teaching of calculus and programming have evolved to make them more effective and widen their audience, the teaching materials for software architecture need to improve. As training materials become more mature, the pool of developers who are able to use software architecture should also grow.

7. Acknowledgements

Great thanks to those on the architecture group and application team who inspired this report and to Jim Herbsleb, Mary Shaw, the participants in her Wordwright course, Patrick Riley, and Marianne Berkovich who provided extensive improvements to the report argument and structure. The author wishes to acknowledge support through the High Dependability Computing Program from NASA Ames cooperative agreement NCC-2-1298.

References

[1] P. Bucci, T. J. Long, and B. W. Weide. Teaching software architecture principles in cs1/cs2. Proceedings 3rd International Software Architecture Workshop, 1998.

[2] J. Cheesman and J. Daniels. UML Components: A Simple Process for Specifying Component-Based Software. The Component Software Series. Addison-Wesley, 2000.

[3] P. Clements and L. M. Northrop. Software Product Lines: Practices and Patterns. Addison-Wesley, 2001.

[4] D. D'Souza and A. Wills. Objects, Components, and Frameworks With Uml: The Catalysis Approach. Addison-Wesley Object Technology Series. Addison-Wesley, 1998.

[5] M. Fowler and K. Scott. UML Distilled: A Brief Guide to the Standard Object Modeling Language. Addison-Westley, 1999.

[6] D. Garlan, R. Monroe, and D. Wile. Acme: An architecture description interchange language, 1997.

[7] D. Garlan, M. Shaw, C. Okasaki, C. Scott, and R. Swonger. Experience with a course on architectures for software systems. Proceedings of the Sixth SEI Conference on Software Engineering Education, 1992.

[8] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering, 26(1), January 2000.

[9] J. Mukerji. Model driven architecture. Technical report, Object Management Group, 2001.

[10] M. Shaw and D. Garlan. Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall Publishing, 1996.

[11] I. Sun Microsystems. Java 2 Platform, Enterprise Edition Specification, Version 1.3, August 2001.