Conducting an Architecture Group in a Multi-team Agile Environment

Mauricio José de Oliveira De Diana[1], **Fabio Kon**[1], **Marco Aurélio Gerosa**[1]

[1]Department of Computer Science – University of São Paulo (USP)
São Paulo – SP – Brazil

{mdediana,kon,gerosa}@ime.usp.br

Abstract. *An important agile principle is that the best designs and architectures are emergent. It is completely feasible in low-complexity systems and / or in a single development team context. However, when systems grow, development is usually split among many teams. When it happens, implementing agile principles that lead to emergent architecture becomes harder. Although there is discussion on scaling agile in general, it is not clear how to deal specifically with software architecture in a multi-team scenario. This article presents an experience report on how a medium-sized company created an architecture group to tackle this problem, and how this group adopted the same agile principles and practices that development teams use in its arrangement and operation.*

1. Introduction

In Agile Methods, the design of an application starts as simple as possible and incrementally evolves from that moment on. Practices such as automated testing, continuous integration, and refactoring together with simplicity principles such as "do the simplest thing that could possibly work" and "You Aren't Going to Need It" (YAGNI) [Fowler 2001] are the foundation of emergent design, an important agile principle [Beck et al. 2001].

In a single-team scenario, an emergent architecture is feasible. However, when development involves a bigger system composed of sub-systems built by several teams, systems architectures become more complex. In such a scenario, the simplest solution for one team may not be so simple for other teams. In this case, the lack of a big picture may jeopardize the whole system conceptual and architectural integrity [Brooks 1975, Bass et al. 2003]. When this vision does not exist, it is hard for those independent teams to find adequate solutions to their architectural issues just by talking to each other and expecting that the whole system architecture will emerge.

In this situation, the first approach that may come to mind is to name a chief architect or create an architecture board, responsible for defining and enforcing the whole system's architecture. The main drawback of this approach is that it creates the necessary conditions to the rise of an anti-pattern observed by Phillipe Kruchten, Ivory Tower architects – architects disconnected from the development day-to-day reality [Kruchten 2008]. This anti-pattern can hurt teams' self-organization, teams' motivation, and architecture emergence, all cornerstones of Agile Methods.

We addressed the problem of handling architecture in a multi-team agile environment in a medium-sized company by creating an architecture group composed of team representatives. To manage the group, we adopted the same agile principles and practices

that work for development. In this article, we show how the group is organized, what it delivers, how it operates, and the lessons we have learned. We believe this approach may be useful for other multi-team organizations facing the challenge of balancing software architecture concerns and agility.

Section 2 describes the scenario which led to the creation of the architecture group. Section 3 explains the group's organization and lists its objectives. Section 4 describes the expected group's deliverables, while section 5 describes the group's practices and operation. Section 6 presents alternative approaches on architecture and multi-team agile. Section 7 concludes this article reporting the main lessons learned.

2. Scenario

Locaweb is a 10-year-old Brazilian web hosting provider, the leader in its market with 22% market share and an annual growth of about 40% over the last few years. The company has around 100 developers in its IT department, who produce applications such as control panels and account administration tools for the company's customers and operational support systems such as provisioning and billing. For many years, the company used an ad hoc software development method or, in better terms, no method at all. Developers were allocated and deallocated to tasks as needed. There was no notion of teams nor projects.

The lack of methodology was not a crucial issue while the company was still small. But as the company grew, so did the complexity of its products and back-end systems. Slipped schedules and quality problems in the IT department became more frequent. Systems maintenance costs rose steadily due to bad design and workarounds. Big Balls of Mud [Foote and Yoder 1999] were common. This situation directly impacted business growth. Good ideas, from a business point of view, were discarded due to their high implementation costs.

By the end of 2007, the company restructured the IT department to fix this situation. After many presentations and discussions, the company decided to adopt agile methodologies, and Scrum was chosen for its ability to organize the development from a managerial point of view, with little dependence on technical issues. It would be harder to implement a method such as Extreme Programming (XP) in the early stages, as most of the codebase was composed of old web scripting technologies such as ASP, which would make essential XP practices, such as automated testing, very hard to adopt.

The department was divided into 13 teams, each one with up to 9 developers and a development manager. These teams and the systems they build are loosely coupled, each team being responsible for everything regarding their respective systems, including the software architecture – we stuck to XP's practice of Whole Team as much as possible [Beck 2005]. Most of these systems are constantly evolving as the company's product portfolio and the business rules change. Although the teams must work as independent as possible, some coordination between them is necessary in crucial situations such as integration with central provisioning and billing systems, which is a must for virtually every system. At first, to decide on issues affecting the whole department, a department board composed of development managers, the product management director, and the CTO was created.

Agile practices usually need to be adapted to the context where they are being

used; what perfectly works for one team may not work for another. Therefore, during the following year, each team experimented with different practices and learned what worked best for them. Each team shared the lessons learned with each other, with the department board serving as the formal channel to exchange experiences. Team independence and autonomy grew considerably as the process improved. Teams were truly becoming Whole Teams, with each responsible for system development from inception to deployment, and from maintenance to operation. In other words, they had total control of their own work.

That environment helped to resolve many problems for each team, including architectural ones. Although individual subsystems improved, the architecture of the whole system, composed of each subsystem and their interactions and interfaces, was still undefined and unmanaged. For instance, most systems communicated with others, usually through RPC. But there were no constraints on what protocols the systems should use. Consequently, a team could build a system using W3C Web Service standards and another team would use an XML-RPC based protocol they created. Any system that needed to talk to both would have to implement both communication mechanisms. Such duplication is wasteful, since implementation costs rise and no real value is added. Furthermore, this approach is error-prone as each new protocol increases system complexity. Each team embraced the XP value of Simplicity [Beck 2005], however what would be the simplest solution for a team often imposed complexity upon the others.

In a scenario with only a few teams, it is relatively easy for each to know each other's particularities and to work out architectural issues as they arise. Naturally, the agile principle that states "the best architectures, requirements, and designs emerge from self-organizing teams" [Beck et al. 2001] would be the most appropriate approach. But with 13 teams in place, the company noticed that communication, and consequently decision making, became cumbersome. It was even more difficult because the teams did not share the same technical background. The company presents a wide range of platforms and programming languages, and teams specializes in one or two of them (e.g., Java or .NET). Self-organization inside each team worked well, but the same could not be said at the department level.

3. The Architecture Group

The company needed to find an approach to balance architectural issues and team agility. The first attempt was to centralize cross-system architectural decisions via the department board. However, the board was composed of managers who, despite having some technical background, had little technical responsibility on their teams. They usually played the role of product owner or project manager. The most technically skilled member on many teams were not necessarily the manager.

In light of this, the company created the Architecture Group, composed of each team's most skilled members, appointed by their respective development managers, and a leader responsible for facilitation, not decision making. The first author of this article has been the architecture group leader since its inception. Due to the difficulties with large scale self-organization, we believe this approach provides an effective way to handle architectural concerns in an environment with many teams building interdependent systems. We believe this approach stands valid while the architecture group size is at most twenty members. If the group grows beyond that, communication problems will likely arise, and

a scale-up strategy may be necessary [Lindvall et al. 2002].

According to Bass et al. [Bass et al. 2003], "the software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them." This definition is very important for the group to understand the relationship between each subsystem architecture and the whole system architecture, and consequently understand how the group actions must take place. The group also sees value in Martin Fowler's more relaxed definition [Fowler 2003] that states that architecture is the set of "things that people perceive as hard to change." Such a broad definition helps the group tackle important issues that may fall outside more formal definitions. Linked to Fowler's definition, another important concept is the difference between an emergent and an evolutionary architecture [Ford 2009]. Since the process of creating the architecture involves addressing what will be hard to change later, to depend on emergence to get there is risky. But it does not mean that all architectural decisions must be made and implemented from the beginning of a project. If a team constantly pays attention to the architecture, it can be implemented incrementally, evolving over time as needed.

The group is mainly concerned with three topics, namely architectural management, development quality, and technical debt. We explain them in the next subsections.

3.1. Architectural management

The architecture group cares about the whole system architecture. From the group's point of view, the subsystems built by each team are the software elements from the Bass et al. definition. This vision bears strong resemblance to the Architecture Team pattern [Coplien and Harrison 2004] defined by James Coplien and Neil Harrison, where "the architecture team's task is to create a high-level partitioning. Much architectural work remains to be completed at lower levels." In this respect, the architecture group influences each team at the boundaries, including issues such as systems integration mechanisms, operational support systems, and quality attributes that must be met by any system. The group does not make, review, nor approve decisions on specific systems architectures. Each team is still responsible for its systems architectures. In reference to Fowler's classification [Fowler 2003], the group is closer to the way the Architectus Oryzus works (whose "most noticeable part of the work is the intense collaboration") than to the Architectus Reloadus (who "is the person who makes all the important decisions"). Fowler states that "an architect's value is inversely proportional to the number of decisions he or she makes." This attitude is aligned with the agile concept of an empowered team, where the developers know the best course of action in each context (something confirmed by our observations).

It is crucial to recognize that it is difficult to make good decisions when detached from day-to-day reality. This was the main reason for creating an architecture group composed of members from all teams instead of creating a separate architecture team, a Community of Practice as advised by Craig Larman and Bas Vode [Larman and Vode 2010]. In describing what a Core Architecture Team is, Scott Ambler says that creating such a group "helps to increase the chance that each subteam learns and follows the architecture as well as increases the chance that the core architecture team will not ignore portions of the system" [Ambler 2002].

3.2. Development quality

Even though the architecture group is mainly concerned with architectural issues, it has also a stake in each team's development processes. As such, its second goal is to define development quality standards. In a distributed systems environment, the lack of quality in a specific system can leak and affect other systems. For example, if a system is poorly tested, it will present more defects, which in turn decreases its reliability. Therefore, the group is concerned with quality in all subsystems development. An effective way to improve quality is the systematic use of software development best practices and testing tools at all levels where validation is necessary. Therefore, the group demands actions from teams regarding testing coverage, code standards, and continuous integration, for example. A software development quality group may be a more suitable way to handle development quality issues than the architecture group. At the time of this writing the department is investigating creating such a group, but until it is completely functional, the architecture group undertakes this responsibility.

3.3. Technical debts

At the time of this writing, many teams struggle with technical debts in their systems. Technical debt is a metaphor to show the costs involved in bad design and/or implementation [Fowler 2004]. The idea is that whenever developers neglect the design of their system, they incur in technical debt. In financial debts, there is interest payment. The same happens with technical debts – the longer it takes for the developers to correct the results of bad decision making from the past, the more interest they will have to pay, and the more expensive the correction will become.

Although paying down technical debts is each team's responsibility, it is common for legacy systems to impede or even prevent the appropriate architectural decisions. Because of this, the group set a temporary goal of supporting teams in paying down technical debts. The group can be especially helpful in situations where specific architectural debts affect many systems. For example, high coupling between systems by different teams makes it difficult for them to maintain and evolve those systems. In such cases, the group helps coordinate efforts to cleanly separate the systems, usually by designing strategies for legacy migration that concerns more than one system. In addition, this help can take the form of informal consultancy services that members offer to each other.

Upon recognizing that much technical debt was the direct result of lack of knowledge and experience, the department implemented many training and knowledge sharing initiatives. By helping its developers to become more knowledgeable, the company expects them to both attack existing debts and to build quality into new systems from the ground up in order to avoid new debts. Although the architecture group is not responsible for training, it strongly supports many such initiatives.

4. Architecture Group Deliverables

The architecture group does not decide on subsystems architecture nor produce code. Instead, its members discuss what they see as important issues affecting the whole system architecture, analyze ways to address those issues, and request teams to take actions on them. The group basically delivers these requests in three different forms: standards, goals, and definition of shared infrastructure.

4.1. Standards

Standards definition is the preferred way for the group to manage the whole system architecture. The group uses the same principles behind standardization groups such as IETF and the Java Community Process, i.e., it states what should be done, but not how. In other words, the group does not make recommendations regarding implementation details, it just defines the interfaces between the subsystems.

The group asserts the quality of its standards by observing each group's adherence to them. One of the group's key principles is that if developers do not see value in what the group is advocating they will just bypass it, especially in an environment where autonomy is so intense. This is similar to when de facto standards become more popular in an industry than their formal counterparts. Consequently, whenever a standard is not broadly adopted, the group understands that it failed to create something of value. With that in mind, the group tries to make all the standards as lightweight as possible and then let them evolve naturally. For example, when discussing a substitute for the in-house systems integration mechanism, the group adopted RESTful web services, the simplest solution it could find that would meet the requirements.

4.2. Goals

Another important mechanism put in place by the architecture group is quarterly goals. Every three months the group members discuss the most important issues affecting their teams, looking for commonalities. Based on that, the group defines goals to be achieved by all teams with the aim of solving two different problems. The first one is related to architectural integrity. Goals are a good way to know that every team has worked on a common issue. For example, if teams require information for capacity planning, a possible goal may be that every system must have an attached monitoring component to gather operational information. The second problem is related to backlog prioritization. Our experience shows that although product owners see value in architectural development, when it is completely up to them, they may discard architectural work to make room for user stories contemplating new features, which are more valuable from the customer point of view. Furthermore, they rarely have the technical knowledge to evaluate architectural activities, so it is hard for them to prioritize them. The quarterly goals make the architecture group a project stakeholder as any other to each team.

The group maintains an evolving architecture manifesto which lists the requirements every subsystem must conform to. At the end of each quarter, the goals related to quality attributes are added to the manifesto. By doing that, the architecture group is periodically incrementing and standardizing what is expected from all subsystems architecture. Creating a manifesto incrementally instead of trying to completely define it from the first moment is a pragmatic way to assume what is feasible at the moment and to always focus on the priorities.

The quarterly goals are composed of a few specific items, usually sharing a common theme. For example, the goals for Q1 2009 had the theme "build the basic infrastructure needed to support development (continuous integration servers, code coverage tools, etc.) and operations (monitoring and statistics)", and were:

- All projects that are not considered legacy systems must have continuous integration in place.

- All projects not considered legacy must have their testing coverage measured.
- All production systems infrastructure must be monitored by an alert system (e.g., Nagios).
- Statistics about all systems not considered legacy must be harvested and displayed by a graphical information system (e.g., Cacti).
- Each team must prepare a lightweight plan describing its actions regarding their legacy systems.

4.3. Definition of shared infrastructure

Many systems share common needs, especially infrastructural ones. For example, a single sign-on solution is desired so the customers, who access many distinct web applications, do not need to login again every time they switch from one to another. Another example is the definition and creation of a system integration environment where each team can test their systems by running their automated test suite using other subsystems. In these situations, the group defines which team should be responsible for this particular concern. Usually a member from the team which will get the most from it steps in as volunteer to do the job. The team may be helped by developers from other teams while implementing the solution, but it will be the only responsible for the maintenance of the system after its deployment.

5. Conducting the Architecture Group

The architecture group uses agile principles and practices that work for development teams, i.e., concepts such as iterations, backlog prioritization, planning, and review meetings. Since its actions happen across many teams and are perceived in the long term, the group adopts these concepts in another scale – for example, instead of 1 or 2-week iterations, monthly ones; instead of daily meetings, bi-weekly ones. Following are some adaptations that the group has made to other practices as well.

5.1. Self-organization

In an empowered team, all technical decisions are collectively made by the team members. The same happens in the architecture group, its leader basically has coordination and facilitation responsibilities, not decision ones. Decisions are not made by majority either, since the group believes that in many situations one or two group members have special knowledge that others may not have, so all members have to discuss until rough consensus is reached. It is the same principle behind Planning Poker [Cohn 2005]. In very rare situations when consensus is not reached the leader makes a decision so the group does not get stuck in endless discussions. But the group believes these situations actually are "bad smells" (similar to code smells) – after some time, it discusses "what happened that we couldn't agree on something even after discussion?". We have noticed that it can be for many different reasons, lack of knowledge on the subject being the most common.

5.2. Demand management

The group has a backlog open to the whole company in an internal wiki. Anyone, group member or not, can add items to it. The backlog is prioritized by the group's leader after listening to other members and stakeholders ranging from the CTO to product owners. A constant worry is to always be aligned to the business, the group should attack the most relevant issues from the company's point of view. So the group considers any demand coming from the outside a very positive sign that it is on the right track.

5.3. Planning and review

The group runs planning and review meetings, occurring every other week, with different purposes. At the beginning of the month there is a 3-hour meeting, mandatory to all group members, when work-in-progress is reviewed and new work is started. After two weeks, the group meets again in an 1-hour review meeting, just to follow-up ongoing work. Presence in this meeting is optional. Since all members have about 10% of their time dedicated to the architecture group, this frequency is sufficient to keep them and their teams informed. These meetings are also a good moment for teams to share what they have been working on regarding architecture specifically, or general issues affecting architecture. In a way, it works as an informal Scrum of Scrums, but focused on the engineering side of ongoing projects.

5.4. Working groups

The group quickly learned that it is almost impossible to keep long technical discussions productive with around 20 people talking. Moreover, many of the discussed topics need profound analysis and research. To deal with this situation, the group found inspiration in the way IETF manages RFCs [Bradner 1998]. Working groups are typically created to address a specific problem or to produce one or more specific deliverables (a guideline, standards specification, etc.). Working groups are generally expected to be short-lived in nature. After the tasks are completed, the group is disbanded. However, if a working group produces a Proposed or Draft Standard, it frequently becomes dormant rather than disband. When the deliverable produced by the working group is a draft, it is presented at the next review or planning meeting to public appreciation.

Not everything needs a working group to be worked out. Actually, the group solves most issues during the planning meeting. Working groups are created when the group notices that a particular issue will not be solved with a quick discussion or when it takes too long to reach an agreement. The group leader and up to six volunteers comprise the working groups. Normally, someone volunteers for a working group due to two reasons: their team may have a stake in the target issue or they can be particular fond of the subject. For example, it would be expected that a working group formed to address user access policies has members from the core team, the hosting team, and people interested in security. The usual working group's mode of operation is to meet once a week to discuss the issue and divide research and/or development between its members until their next meeting.

6. Related Work on Architecture and Multi-team Agile

While agile methods were seeing broader adoption in industry, their first limitations started to show up. How to apply agile methods away from their "sweet spot" (single collocated teams, greenfield projects, etc) became a common concern for practitioners and academics, and we have seen the growth in literature on how to scale agile as a consequence. Scaling agile usually involves a wide range of concerns, we limit this section to works which deal with architecture in multi-team agile environments.

Dean Leffingwell's approach [Leffingwell 2007] is to organize systems in components, and have the components assigned to teams. From that, an architecture team, formed by senior architects or technical leaders from the teams, defines the initial architecture in the beginning of the project. A prototyping team may also be formed to test the

architecture. The process follows important agile concepts such as working in time-boxed iterations and focusing primarily on implementation rather than just modeling. After the initial phase, work is assigned to development teams. They start by building what Leffingwell named the architectural runaway, which "contains existing or planned infrastructure sufficient to allow incorporation of current and anticipated requirements without excessive refactoring." It is expected that the runaway get extended to accommodate new features. However, development teams must not necessarily be responsible for extending the runaway. If architectural expertise is mainly concentrated in the architecture team, this team may work in advance and prepare the runaway so development teams find the needed infrastructure ready when they start to implement a new feature. Leffingwell's approach seems more suited to deal with scenarios where systems are more tight coupled, teams work closer to each other and there are few ongoing strongly related projects at a time, a scenario different from ours.

Jutta Eckstein, following Frederick Brooks' observations, recognizes that a system's conceptual integrity is lost when a team becomes too large [Eckstein 2004]. To avoid that, she suggests the assignment of a lead architect. This person is responsible for making final architectural decisions, remembering the rationale behind those decisions, and, most importantly, spreading the system's architectural ideas so people have a better understanding of its architecture. Besides the presence of the technical lead, Eckstein suggests what she calls architecture as a service. The term means that an architecture team's work should be guided by, instead of guide, the development teams' work. The architecture team build only what is requested by the development teams that, by their turn, only ask for work related to their respective customers. By doing that, an organization sticks to the YAGNI principle, keeps its architecture simple, and ease the burden on developers to understand the architecture. Eckstein's approach seems a good fit in a more homogeneous and stable environment, where the lead architect can keep enough details of the systems architectures in his head. In our case, the number and diversity of systems, platforms, and technologies are such that it is difficult to have only one person looking at them all. Regarding architecture as a service, we follow Eckstein's advice, in a different way. One could see the group's goals as the group guiding the development teams, but the group's members come from each team exactly for the same reason Eckstein suggests her approach: to keep the group working in issues those teams see as valuable.

Roland Faber presents a case study describing how he has organized an architecture team to act as a service provider [Faber 2010], similar to Eckstein's approach. In his organization, architects are responsible for nonfunctional requirements, while developers are responsible for functionality. The architecture process is divided in two phases, preparation and support. During preparation, an architect specifies system qualities, creates an overview of the architecture, creates prototypes and a system skeleton to be used by developers to test and integrate their work early. In the support phase, an architect is collocated with the development team, and helps the developers implement the application, coding with them. Faber notes this is important because it builds trust between developers and architects and provides valuable feedback for architects about their architectural decisions and frameworks. Faber's approach particularly aims at the architecture of each separated system, but with overall issues such as reuse in sight. In our approach, the main concern of the architecture group is the whole system architecture, and each team is responsible for its system's architecture. Another difference is that instead of working with a sepa-

rated architecture team that collocates its members in teams, we followed the other way around, and formed the group with development team members.

7. Conclusion

The architecture group has been working for almost a year by the time of this writing. It already has presented considerable results, maybe the most important being an unplanned one: the increasing of architecture awareness by all company developers. It is an important achievement, since architecture was seen as unworthy two years ago. Besides, so far many mechanisms have been put in place to improve the quality of both customer products and back-end systems, such as the system integration standard, and an extensive use of testing and continuous integration.

Up to now, we have had spotted difficulties of operating an architecture group that others following this approach can face:

- Sometimes, members have a hard time dealing with conflicts between activities in their teams and in the architecture group, specially regarding agenda. For example, a team planning meeting may be scheduled to occur at the same time as a working group meeting. There is no general rule to deal with those situations, but usually one takes the team activities since most architecture group work is aimed at the long-term, and thus can wait.
- There are very few controversial issues where consensus is unreachable. In these cases, enforcing a decision is better than just skipping the issue.
- In a heterogeneous environment, there are situations when the technical lead must conduct discussions, but he does not have sufficient knowledge about the topic or the details involved to do so. When this is the case, the lead has to admit one's lack of confidence, ask for help and study as much as possible about the problem in hand.

In this article, we presented the practices that have been working for the architecture group. They may be useful for other organizations interested in adopting an architecture group following the approach proposed in this article. Table 1 provides a summary of the practices:

Table 1. Summary of the architecture group's practices.

Practice	Responsible	Objective	Details
Fill the architecture group with members of each team	Managers	Connect the group to the day-to-day reality of the organization	Section 3
Create standards	Architecture group	Address technical issues affecting multiple teams while preserving each team's autonomy	Section 4.1
Create goals	Architecture group	Make multiple teams consistently apply specific practices and standards and / or help balancing architectural related development and new features in teams' backlogs	Section 4.2
Define shared infrastructure	Architecture group	Avoid duplicate efforts when multiple teams share the same infrastructural needs	Section 4.3
Decide on technical issues collectively	Architecture group	Find the better solutions by summing the knowledge and points of view of members coming from different backgrounds and specialties	Section 5.1
Create and prioritize a backlog	Anyone add items, the architecture group leader prioritizes	Organize the architecture group's work and make it visible	Section 5.2
Run planning and review meetings	Architecture group leader organizes, architecture group members and any other interest party attends	Organize the architecture group's work and keep track of work being done at the moment	Section 5.3
Form working groups	Architecture group	Avoid long discussions when current information on technical or business is not sufficient	Section 5.4

When the number of people involved in a system goes much beyond the suggested agile team size, it becomes very difficult to keep the whole system architectural integrity. Our approach recognizes the need to balance architectural integrity with team self-organization and autonomy. Although having a central coordinating group in an agile environment may seem contradictory at first, it makes sense if the group not only respects, but also uses, the agile values, principles and practices guiding the whole organization.

8. Acknowledgements

We thank Locaweb for supporting this work and to Andrew de Andrade for revising preliminary versions of this article. Fabio Kon and Marco Gerosa receive individual grants from CNPq.

References

Ambler, S. W. (2002). *Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process*. John Wiley & Sons.

Bass, L., Clements, P., and Kazman, R. (2003). *Software Architecture in Practice*. Addison-Wesley Longman.

Beck, K. (2005). *Extreme Programming Explained: Embrace Change*. Addison-Wesley Longman.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R., Mellor, S., Schwaber, K., Sutherland, J., and Thomas, D. (2001). Agile manifesto. `http://agilemanifesto.org/`.

Bradner, S. (1998). IETF working group guidelines and procedures, IETF RFC 2418. `http://www.rfc-editor.org/rfc/rfc2418.txt`.

Brooks, F. (1975). *The Mythical Man-Month*. Addison-Wesley.

Cohn, M. (2005). *Agile Estimating and Planning*. Prentice-Hall.

Coplien, J. O. and Harrison, N. B. (2004). *Organizational Patterns of Agile Software Development*. Prentice-Hall.

Eckstein, J. (2004). *Agile Software Development in the Large: Diving Into the Deep*. Dorset House Publishing.

Faber, R. (2010). Architects as service providers. *IEEE Software*, 27(2):33–40.

Foote, B. and Yoder, J. (1999). Big ball of mud. In *Pattern Languages of Program Design 4*. Addison-Wesley Longman.

Ford, N. (2009). Evolutionary architecture and emergent design: Investigating architecture and design. `http://www.ibm.com/developerworks/java/library/j-eaed1/index.html`.

Fowler, M. (2001). Is design dead? In *Extreme Programming Examined*. Addison-Wesley Longman.

Fowler, M. (2003). Who needs an architect? *IEEE Software*, 20(5):11–13.

Fowler, M. (2004). Technical debt. http://martinfowler.com/bliki/TechnicalDebt.html.

Kruchten, P. (2008). What do software architects really do? *The Journal of Systems & Software*, 81(12):2413–2416.

Larman, C. and Vode, B. (2010). *Practices for Scaling Lean & Agile Development*.

Leffingwell, D. (2007). *Scaling Software Agility: Best Practices for Large Enterprises*. Addison-Wesley Professional.

Lindvall, M., Basili, V. R., Boehm, B. W., Costa, P., Dangle, K., Shull, F., Tesoriero, R., Williams, L. A., and Zelkowitz, M. V. (2002). Empirical findings in agile methods. *Lecture Notes in Computer Science*, pages 197–207.