Principles of Agile Architecture

Intentional Architecture in Enterprise-Class Systems

By Dean Leffingwell
with Ryan Martens and Mauricio Zamora

Agile Crosses the Chasm to the Enterprise

The benefits of Agile methods are becoming more obvious and compelling. While the most popular practices were developed and proven in small team environments, the interest and need for using Agile in the enterprise is growing rapidly. That's largely because Agile provides quantifiable, "step-change" improvements in the "big three" software development measures – *quality, productivity and morale*. Confirming Agile's benefits, hundreds of large enterprises, many with more than 1,000 software developers, are adopting the methodology.

Clearly, Agile has reached the early mainstream adopters in the software enterprise. As an industry, we must now prepare for the next challenge that these methods present. In our experience, the ability to successfully scale Agile beyond single teams depends on a number of factors.

In this paper we'll discuss one of the factors, the role of "Intentional Architecture" in the development of enterprise-class systems built using agile methods and techniques. In order to gain a better understanding of this important practice, we'll describe:

- The context and challenges of large-scale system architecture in Agile development
- The need for Intentional Architecture to buttress emerging architecture
- How the traditional systems architect can contribute to Agile teams

There are a number of governing principles teams can apply to the challenge of architecture development in large-scale systems. These governing principles will be covered in the section entitled *Principles of Agile Architecture.*

Challenges of Emergent Architecture at Enterprise Scale

Regarding software architecture, it's interesting to note that it is the "lighter-weight" Agile methods, specifically Scrum and XP, that are seeing the broadest adoption in the enterprise. It's even more interesting to note that these methods provide the least guidance (at least as compared to FDD, DSDM, OpenUP and others) on the topic of architecture in Agile development. And that is one of our challenges. These methods are based on the assumption that architecture emerges as a natural outcome of a rapid iteration cycle, implementation of prioritized value-driven user requests and a continuous re-factoring process.

For developers who have experience building large-scale systems with the extensibility and scalability the market demands, this seems counter to what our experience implies, which is that some amount of architectural planning and governance is necessary to reliably produce and maintain such systems.

So, the challenge is to carry the desirable and qualitative Agile benefits from individual teams to multiple teams so the methods can be used to build ever-larger software systems. Do Agile methods scale? Will teams using the methods be able to build software systems that scale to the levels of robustness and extensibility required by enterprises? The answer to both questions can be "yes," provided we reinvigorate the central role of system architecture.

Refactoring Large-Scale Systems

Refactoring is a case in point. Refactoring is as integral to Agile as the daily standup, unit testing and retrospective. It has to be this way because, with minimal or non-existent "up-front" requirements and design, teams understand they can't possibly get it right the first time (as if we ever could!). Instead, they depend on refactoring skills to quickly tack the system to meet the customer's needs.

However, as a better understanding of market needs emerges, continuously refactoring large-scale, emerging architectures becomes less practical as the size of the system grows. In addition, to improve usability, extensibility, performance and maintenance, most large teams apply component-based approaches. It can be beneficial if these components utilize common behavioral and implementation approaches. The benefits of applying "architectural guidance" to these practices include:

Avoiding large-scale and unnecessary rework. Even minor system-level refactors can cause substantial rework for large numbers of teams, some of whom would otherwise NOT have to refactor their module. It is one thing for one team to refactor their code based on lessons they have learned, it's quite another to require multiple teams to refactor code based on other teams' lessons learned.

Managing the risk of Impact on deployed systems and users. Even the best possible Build Verification Tests (BVT) systems are imperfect, so the threat of introducing a regressive bug in a deployed system is always present. And the cost, risk and impact of the bug increase with scale. Minimizing unnecessary refactoring is a primary risk-mitigation technique.

Improve usability, extensibility, performance and maintenance. Some common, imposed architectural constructs can ease usability, extensibility, performance and maintenance. For example, something as simple as imposing a common presentation design can result in higher-end user satisfaction and, ultimately, additional revenue.

Intentional Architecture for Enterprise-Class Systems

The need for guidance brings us to the role of intentional architecture, an enterprise practice designed to produce and evolve robust system architectures in an Agile fashion. Intentional Architecture has three primary objectives:

1. Leveraging common architectural patterns, design constraints and implementation technologies. These are the decisions that can simplify development, extensibility and usability. Some can be defined up-front (examples: use the existing user activity logging component, do all web GUIs for the new subsystem in PHP). Other techniques emerge during the course of development, and can be leveraged by other teams if we only take the care to do so.

2. Building and maintaining architectural runway. Architectural runway exists when there is sufficient system infrastructure in place to allow incorporation of near-term product backlog without potentially destabilizing refactoring. In most Agile practices, it is sufficient to have about six months of runway, so there is a high probability that backlog items with the highest priority can be reliably committed to the nearest, upcoming releases.

3. Sponsoring Innovation. In many ways, Agile natively fosters innovation by quickly driving solutions to meet real-world user needs. However, Agile can also be held hostage to the "tyranny of the urgent" as rapid iteration cycles and continuous commitment to value delivery may drive teams to avoid risky or innovative experiments. When everyone is committed and accountable to a near-term deliverable, who is scouting the next curve in the road?

Role of the System Architect in the Agile Enterprise

There are substantial benefits when Intentional Architecture is effectively applied, provided that development is not slowed and we don't capitulate to the waterfall design phases of the past.

Historically, Intentional Architecture was a primary function of the system architect. But the most common Agile methods don't define or even support such a role. Since Agile focuses on harnessing the power of the collective team, rather than any one individual, the system architect no longer dictates technical direction.

While these system architects have decades of technical experience, this expertise has most likely taken place outside of the Agile process, and they may not understand the construct of building refactorable code. Indeed, they may view the practice as unnecessary rework, and might not support the Agile model.

System architects may also be concerned about the potential architectural entropy of all the newly empowered and energized Agile teams. They may also have strong opinions about the software development practices teams employ. If we fail to bring these key stakeholders on board to the Agile development paradigm, they could quickly kill the entire initiative.

We want to avoid at all costs a battle between Agile teams and system architects, for there will be no winner in that fight. Therefore, it is definitely in our best interest to include system architects in the Agile process, and their input should be highly valued by the team.

Principles of Agile Architecture

As with all things Agile, we must constantly remind ourselves to return to the first principles, such as the Agile Manifesto and its derivative works. The works contain the founding principles for the methods and best practices that have served us well. They have proven to be remarkably intuitive and durable, and have helped guide many enterprises to large-scale Agile success.

However, while the spirit of the principles guides us, they are largely silent on the topics of enterprise-class systems and architecture in general. That's understandable, as these methods weren't developed in the context of larger systems of systems. But, as we extend the effective techniques to achieve even greater benefits, we have to move to the next level of scale without compromising the principles that have brought Agile success so far.

To help us step up to the next level, we propose a set of governing principles for the development and maintenance of Agile, enterprise-class, intentional architectures. These principles are:

Principle #1 The teams that code the system design the system.
Principle #2 Build the simplest architecture that can possibly work.
Principle #3 When in doubt, code it out.
Principle #4 They build it, they test it.
Principle #5 The bigger the system, the longer the runway.
Principle #6 System architecture is a role collaboration.
Principle #7 There is no monopoly on innovation.

We'll describe each of these principles further in the sections that follow.

#1 The teams that code the system design the system.

This first principle is driven by the Agile Manifesto itself, which states that the best architectures, requirements and designs emerge from self-organizing teams, and another of the primary philosophies of Agile development: Teams themselves are empowered to define, develop and deliver software, and they are held accountable for the results.

From a management perspective, in order for teams to be held accountable, we must allow them to make the decisions required to support that accountability. If not, they will be held accountable for decisions made by others, and that is an ineffective and de-motivating model for team performance. While this seems axiomatic, responsibility in earlier practice was different, as Table 1 (below) shows.

	Pre Agile	Post Agile
Management	• Determines market needs and features • Communicates vision • Product managers determine requirements • Architects determine architecture • Management determines schedule and commits on behalf of team • Accountable for the results	• Determines market needs and features • Communicates vision • Eliminates impediments for the team • Accountable for empowering teams to deliver
Team	• Inherits the plan • Inherits the architecture • Left "holding the bag" and executes on a "best efforts" basis	• Determines the requirements • Determines the architecture • Determines the schedule in terms of iterations and releases • Commits on behalf of themselves • Accountable for the results

***Table 1** – Responsibility and accountability in the pre- and post-Agile world*

This level of responsibility was further compounded by time-sequenced waterfall activities. Typically, there was a single, up-front planning phase intended to accommodate the inherent risk of the project, perform all the necessary design and predict sequenced-task dependencies, running out as far as a year. Because it happened up-front, the planning phase was decoupled from the lessons learned later during implementation. Once the lessons were learned, it was too late to do anything to impact the schedule, except to apologize for the fact that the schedule was not met.

From a technical perspective, architectural decisions are most optimally made by the coders, the technical leads and team-based architects because they are closest to the implementation and often have the best data available to make such a decision.

Moving these responsibilities to the team is a triple win for the enterprise:

1) A more optimum decision is likely to be made
2) Once a decision is made, the team will likely work much harder to make its decision work in the implementation
3) Regardless of what happens, the team is empowered, responsible and accountable for its decisions

#2 Build the simplest architecture that can possibly work.

This principle certainly comes as no surprise, even to those new to Agile, because Agile is famous for its focus on simplicity:

> *"What is the simplest thing that can possibly work?" – attributed to Ward Cunningham*
>
> *"If simplicity is good, we'll leave the system with the simplest design that supports its current functionality." – Kent Beck*
>
> *YAGNI – You Ain't Gonna Need It – an XP mantra*

Does simplicity remain an essential attribute as complexity increases? We believe the answer is yes, and is supported by our experience in building large-scale systems. In most cases, the simplest decisions turned out to be the best over time. Rally's customers also support this principle. For example, Amazon used Agile and organically grew a system to handle 55 million customer accounts. Werner Vogels, Amazon's CTO, said:

> *"The only way to manage a large distributed system is to keep things as simple as possible. Keep things simple by making sure there are no hidden requirements and hidden dependencies in the design. Cut technology to the minimum you need to solve the problem you have. It doesn't help the company to create artificial and unneeded layers of complexity."*

#3 When in doubt, code it out.

Agile, with its highly iterative experience and code-based emphasis, allows developers to simply rely on their coding skills to move efficiently through the decision-making process. This is helpful when selecting a design alternative or a high-impact infrastructure implementation choice. But we may still occasionally find ourselves mired in technical debate.

This principle reminds us that when we have to make a tough choice, we can always turn to a rapid evaluation in code. Fast one- or two-week iterations give us a quick project cadence, and the demos at the end of the iteration provide objective evidence of results.

Inherent visibility of the Agile model also allows all impacted stakeholders to see the real-time, in process reasoning and experimental results. In addition, Principle #1 reminds us that if a design alternative can't be coded and evaluated within a few iterations, it probably isn't the simplest thing. In practice, cases where a decision wasn't fairly obvious after a few short design spikes are rare.

Another lesson learned from the Amazon Architecture project:

> *"Use measurement and objective debate to separate the good from the bad. …this is the aspect of Amazon that strikes me as uniquely different … Their deep seated ethic is to expose real customers to a choice and see which one works best and to make decisions based on those tests. …. calls this getting rid of the influence of the HiPPOs, the highest paid people in the room. This is done with techniques like A/B testing and Web Analytics. If you have a question about what you should do, code it up, let people use it, and see which alternative gives you the results you want."*

#4 They build it, they test it.

Agile is renowned for "forcing" testing early in the lifecycle of the development process. Many Agile thought leaders implemented unit testing and acceptance testing frameworks into the base Agile technical practices. Concurrent testing is a cornerstone practice of Agile, and is a primary reason why quality is significantly higher in Agile, without sacrificing developer productivity.

Philippe Kruchten once described architecture as "take away everything in the system you don't need to describe how it works, and what you have left is its architecture". So how do you test architecture? You simply test how the system works without the details. In other words, testing architecture involves testing the system's ability to meet its large-scale functional, operational, performance and reliability requirements. To do this, teams must build an infrastructure that enables testing.

Because testing represents complexity at its highest level, the team that codes the system must be the team that determines how to test the system. With the complexity of today's automation frameworks, developers are likely to be directly involved in applying testing automation. It is the responsibility of the development teams to develop, test and maintain a system-testing framework that continually assess the system's ability to meet its architectural and functional requirements. This responsibility cannot be given to any other testing resource or outsourced function.

#5 The bigger the system, the longer the runway.

At the release level (internal or external), value delivery focuses on delivering the features customers need. The ability to deliver planned functionality predictably in a near-term (60-120 days) release is a hallmark of mature Agile teams. That ability, in turn, allows Agile enterprises to communicate expectations to customers, whose businesses depend on new software releases. One of the key benefits of Agile is that the team meets its commitments, and the software actually works.

But even experienced Agile teams occasionally have trouble completing iterations. In general, that can be acceptable, as a team that reliably completes 100 percent of the stories may not be stretching enough to meet the demands of the marketplace. Furthermore, so long as the team is able to self-correct effectively, it also encourages a level of acceptable risk taking.

However, when we see an iteration that is missed badly (<50 percent of story completion and failure to deliver even the highest-priority stories), then the release itself may be at risk. In those cases, there is typically a serious architectural work at play, and the team simply underestimated the time it would take for a significant refactor or to lay in a new foundation. This leads us to the conclusion that an Agile team's ability to meet value delivery commitments is far more reliable when the foundation for the new features is already in place.

This is why we stress the need for the continuous build out of "architectural runway" – system infrastructure that must be in place to deliver features on the product roadmap – as a mechanism for decreasing the risk of missed commitments.

For smaller teams, infrastructure to support a single iteration or release cycle may be all the runway that's needed. It may be much more efficient for those teams to be wrong initially, and then refactor the application, than it is to invest time up front trying to discover the undiscoverable. For larger teams and systems, however, building and refactoring infrastructure takes longer than a single short release cycle. Because of this, it's necessary to build most features for a particular release on existing infrastructure. This requires some additional foresight and investment in Intentional Architecture, or more runway. Without additional runway, the team won't be able to reliably "land" each release on schedule.

While building and maintaining architectural runway is outside the scope of this paper (see Chapter 18 of Scaling Software Agility), a few tips may be beneficial:

- Intentional Architecture takes time. A best-case scenario is that it's identified in release planning, built in the current release cycle and then consumed in following iterations and releases. If this doesn't work, you could be faced with two possibilities: 1) defer the features that require it until the next release or 2) take the risk of building and consuming the changes during the course of the newly committed release. While a third alternative, pushing the teams to take on the effort within the currently scoped release, sounds tempting, it will likely lead to missed release commitments, and it may negate the team "buy-in" principles.

- To help assure the teams buy-in to the changes, you may want to create a virtual team comprised of the core infrastructure team (if one exists) and a few other iteration team members.

In any case, the bigger the system is, the more attention your teams will need to pay to laying in Intentional Architecture.

#6 Systems architecture is role collaboration.

So far we've focused on the central role the teams play in designing and architecting systems of scale. Now we'll touch on the role of the senior, domain-experienced system architects we mentioned earlier.

Fortunately, enterprise agility is not a zero sum game and there is room for all who can contribute to the best possible technical solutions. After all, we are building systems of enormous complexity (even when we keep them as simple as possible), so why not leverage the skills of those external team members who have the experience to match the challenges the teams face?

How do we incorporate them into our team-centric Agile model? We should remember the parallels that exist between organizations and the systems architecture that organizations create. Conway's law states that: "Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations." While this law may seem to imply a negative, that needn't be the case. For example, in object-oriented systems development, we design systems of collaborating objects, structured around well-defined interfaces, which work together to create an output that is greater than the sum of its parts. The same approach can be applied when working together to develop a system architecture (see Figure 1).

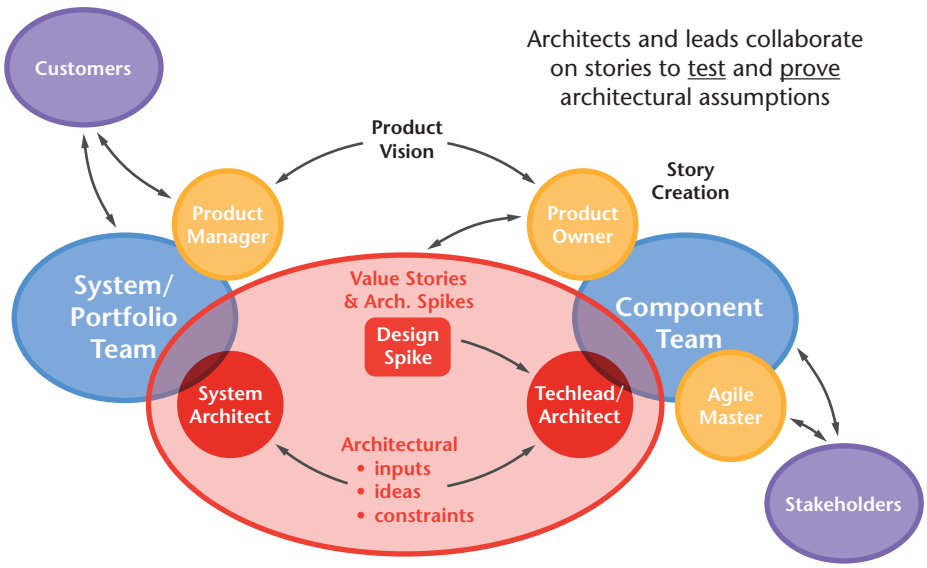

***Figure 1** – Systems architecture is a role collaboration*

In this illustration, system architecture is a "role collaboration" between the system architect and technical leads of the component teams that write the code. These system-level teammates work together with the component teams to decide what the architecture will look like. And when-in-doubt, they-code-it-out with a series of design spikes inside iteration or release boundaries. With the support of the product owner, design spikes are mixed in the backlog, based on the priorities the team feels is appropriate. This is one of the reasons effective product owners often have a high degree of technical experience. Using this model, a consensus emerges as to how to build the system that is about to be deployed.

As an example, let's look at a Single-Sign-On (SSO) architectural consideration from real world experience. In this application, the component teams needed to implement SSO from their system to an external web service. The component teams had little experience in web SSO, having had no prior need for it in their system. The system architect had broader experience and, perhaps more importantly, had been researching SSO options prior to the release planning session where the discussion was initiated. There were many options available, and the system architect already formed an opinion based on prior investigation. But even with that input, a lively discussion ensued.

Who decides, the team closest to the implementation or the more experienced architect who is removed from the implementation? The answer must come through collaboration and agreement between the parties, supported as necessary by evaluation matrices (footprint, recurring cost, level of security, etc.) and, when necessary, a series of design spikes to test each proposed solution. If a design spike or two couldn't prove the feasibility of one choice or another, then it's unlikely the solution chosen is the-simplest-architecture-that-can-possibly-work.

While the experienced team at the top may have some of the answers, localizing the decision process eliminates the buy-in and ideation that is crucial to Agile. This collaborative approach alleviates the "system architect vs. component team" battle because the architecture evolves with the consensus of both parties.

In addition, not leveraging the thoughts and strengths of experienced people throughout the Agile release train sends the wrong message, that team members must "grow" into elite architects to progress in their career. The goal is to create highly efficient teams at all levels because Agile leverages the power of everyone who can contribute.

In summary, if we want to beat the competition to market, we must work together, maximize the contributions of all team players and master the complex systems we are crafting today. To reach these goals, it's crucial to engage system architects in the solution.

#7 There is no monopoly on innovation.

"Inertia is the residue of past innovation efforts. Left unmanaged it consumes the resources required to fund next-generation innovation." — Geoffrey Moore

In the last principle, we described the need for teams to have sufficient architectural runway to reliably "land" near-term iterations and releases. This is the upside of inertia: Enough is known from prior experience and development to allow the team to provide reliable and consistent value delivery. Indeed, with Agile, we have strong technical and project management mechanisms, and some architectural runway, in place to help us stay focused on this sole purpose. Agile practices provide a disciplined, production-like ability to reliably meet commitments and rapidly evolve a system to meet existing customer requirements.

But there is a downside as well. If we are not careful, the "tyranny of the urgent" may keep us focused only on near-term deliverables. So where does the innovation come from in such a model? Mature Agilists put processes in place to assure that innovation is not just incremental and near term.

Some of the innovation comes from empowering system architects as part of our advanced guard. They can be exploring new technologies, patterns and techniques that will help us innovate. But ours is a team-centric model, so we don't rely on architects as the sole source of such innovation. In fact, the team-centric model can foster innovation at an even greater pace than that generally seen in traditional software organizations. That's because true innovators innovate at all stages of their career, and the team-centric model enables these people to flourish and contribute beyond what their level of experience may imply.

One way to foster iteration at the team level is by judicious backlog management that includes spikes for refactoring, design and exploration of new ideas. This can work quite well, but even more explicit models have been put into use. For example, at Rally Software Development, where they have been building their SaaS Agile Product management solution in a highly Agile fashion for four years (with rarely a missed or delayed release commitment), they have evolved to an advanced development cadence as illustrated in Figure 2 below.

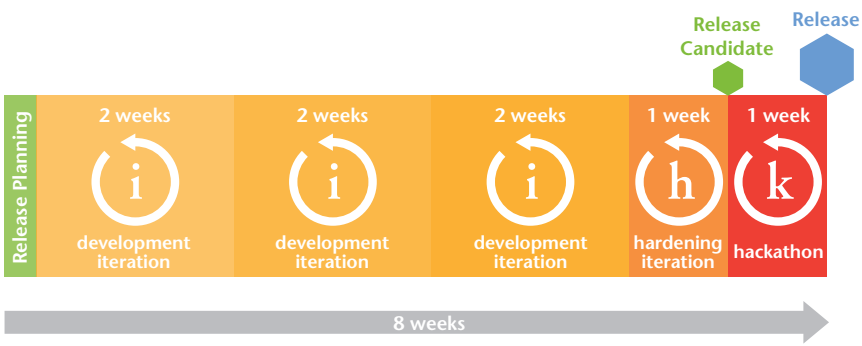

***Figure 2** – An iteration and release cadence with one innovation "hackthon" per release*

This figure illustrates a standard release cycle, where:

"i" is a standard development iteration, providing new functionality for an upcoming release

"h" is a one week hardening iteration, to eliminate technical debt and assure quality requirements meet the release-to-manufacturing criteria

"k" is a "hackathon".

The hackathon is designed to foster innovation at the team level. The rules of the hackathon are simple: Any team member can explore any technology area in any way they want, as long as there is some correlation to the company's mission. This gives the team some mental down time to reflect, think and experiment outside of the everyday rigor and pressures of the iteration and release cycle.

Conclusion

In this paper, we've described the critical role that Intentional Architecture provides in helping teams of teams build reliable, extensible, enterprise-class systems in an Agile manner. To implement this practice, we've empowered the role of the system architect as an integral part of "what makes an Agile team a team." In order to guide the process further, we've proposed a set of guiding principles, Seven Principles of Agile Architecture, intended to be quintessentially Agile, and yet provide guidance in an area where some Agile practices have remained largely silent. We are putting these principles to work in our many project environments and, so far, to good effect. Of course, we are also interested in your opinions and experiences in the journey to build your Agile enterprise. So please join our collaboration at http://scalingsoftwareagility.wordpress.com or http://Agilecommons.org/.

About the Authors

Dean Leffingwell is an entrepreneur, software executive, consultant and technical author who provides product strategy and enterprise-scale agility coaching to large software enterprises.

Mr. Leffingwell has served as chief methodologist to Rally Software and formerly served as Vice President of Rational Software, now IBM's Rational Division, where he was responsible for the RUP. He was also the founder and CEO of Requisite, Inc., makers of RequisitePro. His latest book is Scaling Software Agility: Best Practices for Large Enterprises, published by Addison-Wesley, and is also the lead author of the text Managing Software Requirements. He can be reached through his blog at http://www.scalingsoftwareagility.wordpress.com

Ryan Martens is the founder & Chief Technology Officer of Rally Software and an expert in assisting organizations in transitioning from traditional development processes to more Agile techniques.

Before founding Rally Software Development - his fourth software start-up - Mr. Martens directed the corporate adoption of Internet technologies within Qwest Communications, and then moved on to co-found Avitek, a Boulder-based custom software development firm where he served as Vice President of Marketing & Business Development. Mr. Martens successful efforts at Avitek culminated in an acquisition by BEA Systems in 1999. At BEA, Mr. Martens served as Director of Product Management for the eCommerce applications division and he was instrumental in growing that division to more than $50 million in revenue within its first twelve months.

Mauricio Zamora is an Executive Director at CSG Systems who is passionate about helping others leverage agility to deliver large scale and complex software architectures.

Mr. Zamora cofounded Telution, a company focused on BSS and OSS software within the Telecom and Cable industries. In 2006, Telution was acquired by CSG Systems for their Product Catalog and Business Services assets. He currently leads the team responsible for implementing an Agile release train consisting of 10+ iteration teams with over 100 software practitioners who run in concurrent 2 week iterations. Mr. Zamora frequently posts on the blog: http://agilejuice.wordpress.com

Bibliography

Leffingwell, Dean. 2007. *Scaling Software Agility: Best Practices for Large Enterprises*. Boston, MA. Addison-Wesley

Note:
This whitepaper evolved from a series of posts on Dean Leffingwell's blog at: scalingsoftwareagility.wordpress.com
As of this writing, the original posts are still there, along with various reader comments and contributions.

In Part III of Leffingwell's book, *Scaling Software Agility: Best Practices for Large Enterprises*, the lead author of this article described a number of these practices.