Agile–Architecture Interactions

James Madison

Architects can bring agile and architecture practices together to pragmatically balance business and architectural priorities while delivering both with agility.

Agile development starts to build before the outcome is fully understood, adjusts designs and plans as empirical knowledge is gained while building, trusts the judgment of those closest to the problem, and encourages continual collaboration with the ultimate consumers. Architecture establishes a technology stack, creates design patterns, enhances quality attributes, and communicates to all interested parties. The combination of these two spaces is agile architecture—an approach that uses agile techniques to drive toward good architecture. Successful agile architecture requires an architect who understands agile development, interacts with the team at well-defined points, influences them using critical skills easily adapted from architectural experience with other approaches, and applies architectural functions that are independent of project methodology.

Architectural Interaction Points

Figure 1 shows a simplified hybrid of scrum,[1] Extreme Programming,[2] and sequential project management that I've found effective for guiding architectural work on 14 agile projects over the last eight years. Table 1 briefly describes Figure 1's elements; the architectural functions are ones an architect typically performs on projects, although the list isn't exhaustive. Table 2 shows how architectural functions intersect with interaction points and an architect's main concerns at that intersection. Taken together, the three categories and their four items create a framework useful for understanding and guiding agile architecture that's extensible by adding more categories or items on the basis of other priorities or preferences.

Up-Front Planning

Each architectural function begins in an agile project with up-front planning, much as it does in any project, regardless of methodology. The architect

- makes major hardware and software decisions, mostly by using existing corporate standards, perhaps by lining up proofs-of-concept for new products;
- establishes important design patterns at broad[3] and detailed[4] levels;
- identifies large opportunities for component or service reuse;
- generates high-level diagrams;
- outlines quality attributes,[5] both technical and business, and baselines their trade-offs;[6] and
- establishes communication channels by meeting with stakeholders to understand their concerns and share the general technical direction with them.

Although much of this is similar to the activities of a nonagile approach, up-front architectural work
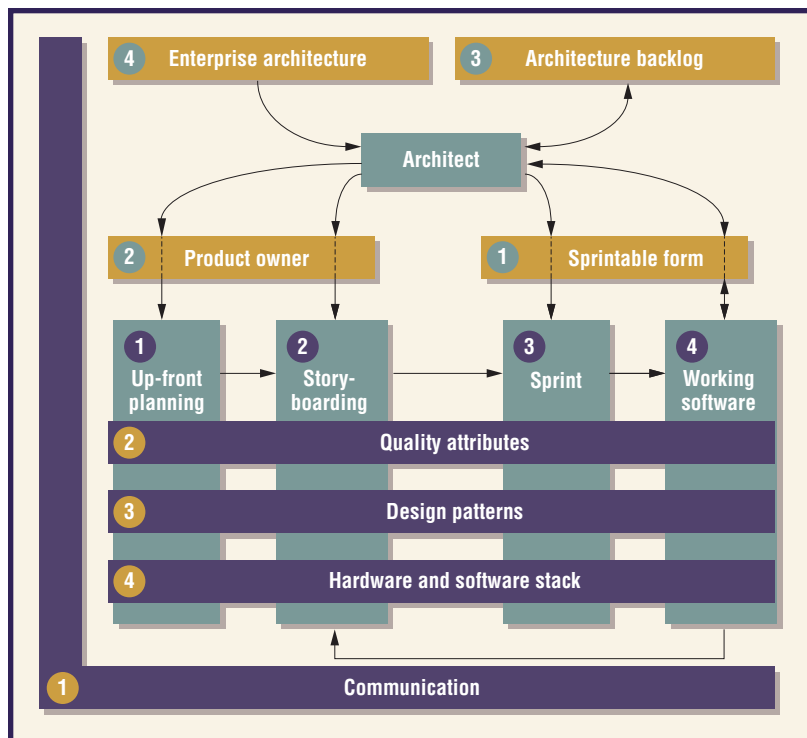
Figure 1. A hybrid framework for agile architecture work. The architect's involvement during project execution helps achieve project objectives. Table 1 further explains the framework's elements: interaction points (green), critical skills (gold), and architectural functions (purple).

empirical knowledge gathered by all participants while building the system will make better options more evident.[7] An architect does well to never lock in a solution too early, of course, but avoiding this trap is particularly valuable with agile development. Agile's use of iterations, construction of working software, and encouragement of collaboration produces a feedback loop that provides tremendous opportunity for all participants to find better solutions later that they couldn't have understood sooner.[8]

For example, on a data warehouse project, the question arose of whether to feed data directly to another area or to build an intermediate data mart. A direct feed was more complex, resulting in lower maintainability and operational efficiency. However, a data mart used more space, resulting in higher cost for the life of the system. The architects observed that either option could meet the business objectives, that we couldn't determine the better design, and that we had confidence in the team to make the right decision within the architectural bounds. As the project executed, the answer became self-evident, the team quantified it nicely, and the architects signed off on it. Defining ranges and bounds is more agile than stating specific solutions, but the architects must still define the ranges and bounds up front.

Storyboarding and Backlogs
Up-front planning moves quickly into storyboarding and building the product/sprint backlogs, with the architect being a key stakeholder. The architect must attend the early storyboarding sessions and contribute architectural user stories that have sig-

with agile development includes a subtle but important difference. The architectural direction should include a range of options rather than a specific solution. A looser set of architectural possibilities is acceptable based on the agile assumption that the

Table 1
Explanation of the elements in a hybrid framework

Category	Item	Description
Interaction point	1. Up-front planning	Setting the architectural direction in much the same way as sequential projects
	2. Storyboarding	Structuring the business need and architectural work, and getting everyone on board
	3. Sprint	Building the functionality as part of the team when direct participation is valuable
	4. Working software	Reviewing what's actually delivered to measure the architectural state
Critical skill	1. Sprintable form	Breaking architectural work into small, measurable units
	2. Product owner	Quantifying the architecture in terms of clear business value
	3. Architecture backlog	Tracking architectural concerns and balancing them with business priorities
	4. Enterprise architecture	Knowing the larger architectural picture and using each project to advance it
Architectural function	1. Communication	Keeping all stakeholders informed about the architecture's value and state
	2. Quality attributes	Measuring maintainability, scalability, extensibility, and similar "-ilities"
	3. Design patterns	Outlining the structures that give form to implementation work
	4. Hardware and software stack	Choosing appropriate hardware and software for the project

Table 2

Applying architectural functions at agile interaction points

Architectural function	Interaction point			
	Up-front planning	**Storyboarding**	**Sprint**	**Working software**
Communication	■ Understand business objectives. ■ Get input from the technical team. ■ Communicate the general direction to everyone.	■ Actively facilitate storyboarding sessions. ■ Work architectural user stories into the backlog, particularly the types in the three cells immediately below:	■ Attend daily stand-ups. ■ Build functionality as a means of gaining understanding. ■ Mentor and assist as expertise allows.	■ Attend the sprint review. ■ Review documentation. ■ Advocate refactoring for architectural value with the team and product owner.
Quality attributes	■ Set approximate target ranges for attributes. ■ Establish which attributes dominate in trade-offs.	■ Add stories to improve specific attributes, including refactoring.	■ Build attributes into code, explicitly and as a norm for build work. ■ Assist in designing or building to improve attributes.	■ Verify that the delivered solution meets target ranges. ■ Adjust target ranges if build work indicates a need for adjustment.
Design patterns	■ Choose important design patterns. ■ Outline general interactions among significant patterns.	■ Add stories to build design patterns, including refactoring.	■ Solve for detailed design patterns. ■ Assist in building the most critical design patterns.	■ Verify that the delivered design patterns are valid. ■ Adjust design patterns as build work indicates.
Hardware and software stack	■ Reuse the corporate stack. ■ Prototype early to verify assumptions. ■ Plan carefully; hardware and software changes are inherently nonagile.	■ Add stories designed to validate hardware and software.	■ Validate hardware and software selection in early sprints. ■ Change early and quickly if stack needs adjusting.	■ Verify hardware and software by continually delivering business functionality on it. ■ Deploy to other environments routinely.

nificant foundational or directional influence. He or she must also attend the ongoing storyboarding between sprints to contribute architectural user stories that fine-tune the architecture or correct undesirable deviations. The architect must work with the product owner to prioritize these stories with the business user stories and build them in conjunction with business functionality in sprints.

The architect often becomes a driving force in storyboarding on the basis of his or her comprehensive knowledge of both the business and technology. I've found that a good architect is well positioned to draw requirements out of the business in storyboard form, explain technical constraints to the business, and restate business needs in technical terms for the team. As the architect does this, he or she can help all parties succeed while smoothly integrating architectural user stories into the storyboard and product backlogs.

For example, a data-warehousing program sought to achieve a high level of enterprise data integration. The architects advocated using dimensional modeling as the primary approach. They also advocated using the bus matrix as the primary tool for organizing data work because the bus matrix facilitated problem decomposition and work iteration.[9] The business (and most of the techni-

cal community) had never used the bus matrix, so the architects had to provide extensive facilitation in the first storyboarding session. By the third session, the product owners came in with their stories printed out in bus matrix form. By the fifth session, the team expressed concern that success was being judged only by the bus matrix components. So, we had to back off a bit and emphasize the value of less visible work such as reusable code components, solving data quality issues, and getting new tools to work. The approach had clearly gathered its own momentum, but the architects' early facilitation got it started.

Sprint Participation

Writing code is a powerful way to ensure that the architect fully understands the architecture being produced,[10] but we'll assume that the organization derives high value from spreading architects around, reducing their ability to be fully hands-on. Fortunately, agile offers a solution—trust the team. This requires the architect to collaborate heavily with the team during the sprint, understanding the objectives, and helping with challenging design issues.[11] To handle multiple projects in this way, the architect must leave many of the specifics to the team. As long as the architect's review of the work-

ing software continues to indicate high architectural quality, the architect can leave the details to the team members, confident that their combined technical knowledge and proximity to the work will keep things on track. That said, engaging in hands-on implementation can become justified when sprints seem to be going off track, architecturally and otherwise. At such times, the architect becomes a hands-on contributor, collocated with the team, with full accountability to the team for the completion of his or her assigned work.

For example, there's a long-standing question in data warehouse architecture about when to use normalized versus dimensional modeling and to what degree.[12] The architects addressed this dispute early in a particular agile data warehouse project by recommending that both be done in their fullest form for maximum functionality. After several sprints, the project velocity wasn't tracking to the required timeline—a common condition regardless of methodology. To see whether architectural changes could help speed up the project, I participated in a hands-on role for the first time in the fourth sprint. On the basis of both the hands-on work and spirited input from seven experts on two teams, it quickly became apparent that moving all the data through a normalized model to land it in a dimensional model wasn't necessary to meet the business objectives (for this particular project, not necessarily in general). We'd been planning the normalized layer for over a year; then on the basis of this new insight, we dropped it in under 30 days. Extensive discussions with management and architectural governance were needed, but the change was made by the next sprint and gave the project a solid velocity increase.

Working Software

After each sprint, the team and product owner must present the working software in a formal sprint review so that all stakeholders, one of whom is the architect, can observe overall progress and provide feedback. Sprint reviews tend to last only a few hours with many stakeholders vying for conversation time, so the architect should start reviewing the working software several days before the official review. It might still have some work-in-progress aspects, but with a formal sprint end approaching, the software should be stable enough for a meaningful review. Well-run agile projects require the iterative delivery of documentation with the working software, including architectural documentation—undocumented code and system functionality shouldn't be considered working software. Reviewing this documentation as it emerges from each sprint is a useful form of architectural review. What's more important, the architect should review the working software by getting deep into the code and system functionality.

For example, over the past decade, I've accumulated several hundred scripts that automate the architectural analysis of a data warehouse platform or data-processing application. When my teams release working software, I run my scripts. Within minutes, I have reports that thoroughly describe the health of the platform, schema, data model, data quality, and other aspects of the data architecture. Any issues discovered can be addressed in the current sprint or queued in the appropriate backlog. To help the process scale, I offer the teams my scripts so that they can perform automated architectural inspection without me. Inevitably, they have some valuable script that checks something important that I missed. Together we grow the system's architectural quality as we try to one-up each other with the slickest way to automate architectural inspection.

An Agile Architect's Skills

Jumping into these interaction points as an architect can be a turbulent experience. Everyone's busy, developers might view architects with skepticism, and there always seems to be a business priority that justifies bypassing good architecture. Minimizing the turbulence requires many subtle skills that only grueling experience can optimize, but four top the list.

Decomposition into Sprintable Form

Agile development requires the product owner to decompose user stories until they're small enough to be executed in a sprint while still being substantial enough to show business value. Likewise, the technical team decomposes user stories to a form that can be efficiently built within sprints. The architect's contribution to decomposition consists of identifying the boundaries of architectural significance and working with the product owner and technical team to ensure that the overall decomposition of work follows these boundaries. An architecturally significant boundary exists between any two collections of business or technical functionality whose hardware and software, design patterns, or quality attributes are nontrivially different. Consider the two examples in Figure 2.

In the first example, we needed to build an enterprise Web service for third-party data using service-oriented architecture (SOA) practices. The service project team used a nine-sprint approach structured around the three major areas of techni-
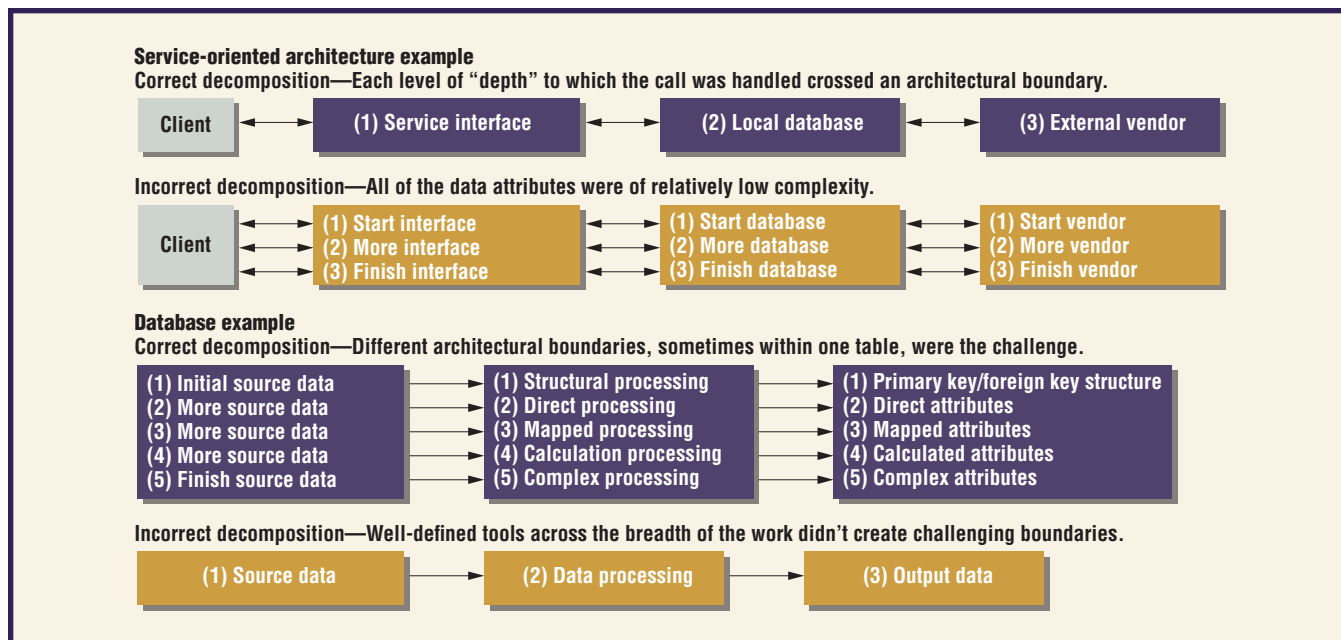
Figure 2. Everyone contributes to decomposing user stories to sprintable form. The architect contributes by leveraging the boundaries of architectural significance, as shown in these examples. The numbers represent sprints or clusters of sprints, depending on the amount of work.

cal functionality—the service interface, the persistence layer, and the external data retrieval. In the first few sprints, the team published the service interface. A client call to the service returned only one hard-coded record, but the transaction was via a fully functional service call with a well-defined contract. Architecturally, this tackled Java, Web services standards, XML, and calling patterns while giving the client system a record for building screens to show the business. In the second cluster of sprints, the team enabled the service to returned about 100 records from the local database but not from the external vendor. This tackled the database environment, data model, and object-relational mapping layer while showing more cases for business review. In the third sprint cluster, the team made the service call the external vendor. This tackled the firewall issues, vendor data format, and latency requirements. From early on, the growing functionality of the service provided a concrete measure of progress based on working software, letting the team focus on a narrow set of technical challenges while giving the business visible value.

In the second example, we needed to deliver a large data warehouse environment. From a business view, data warehouse deliverables lend themselves nicely to decomposition at the level of data subjects, which tend to map nicely to well-defined table structures. But from a technical view, attributes within a table can have significant archi-

tectural differences. For example, premiums and losses are basic insurance information that come straight from source systems, but rerated premiums and developed losses are complex calculations that can warrant entire systems unto themselves.[13] From a business view, premiums are one category and losses are another. From an architectural view, basic data attributes are one category and complex calculations are another. To balance these differences, the team decomposed the work according to complexity, allowing the basic attributes to be delivered quickly while progressively building more complex attributes more slowly.

For each of these examples, and in general,[14] the team needed to give as much consideration as possible to making business-centric decomposition the primary approach. But for efficient project delivery, the architectural boundaries must sometimes prevail because iterating across architectural boundaries can open too many simultaneous challenges, causing risk to the project. If we had tried to decompose the problem across the data in the SOA example the way we did in the data warehouse problem—for example, by moving one-ninth of the attributes end-to-end across nine sprints—the team would have had to address many new technologies at once. This would have caused great trouble, even if the business preferred to see live data much earlier than it did.

Likewise, if we had tried in the data warehouse example to decompose the problem across technology layers the way we did in the SOA problem, it would have slowed basic attributes to the speed of the difficult attributes. This would have caused severe underdelivery, even if the business preferred to see its most complex attributes as soon as possible. The inability of these two examples to use each other's decomposition approach also shows that the decomposition must match the nature of the deliverable.

Advocacy with the Product Owner

The path from decomposed problem to working software runs through the product owner, requiring the architect to promote the architectural work's value with that person. The two most critical aspects of this relate to creating and refactoring the system design and stating the value of quality attributes in terms of business value. Each requires time from the team that will compete with business functionality. If the product owner doesn't understand the architectural work's value, the work will continually get low priority in the product backlog and result in an inferior architecture.

Fortunately, nearly all design work contributes directly to quality attributes and nearly all quality attribute improvements translate into business value. Maintainability results in business functionality being built faster in later sprints with quicker enhancement turnaround for the life of the system, resulting in faster speed to market. Scalability results in the system still delivering fast performance when it encounters a huge spike at the peak of an important marketing campaign, preventing the loss of business at critical times. And so on. Naturally, this connection from good architecture to business value isn't unique to agile development. But the power that agile development gives the product owner makes it particularly important to clarify the value frequently, with the most important advocacy taking place in the first few sprints when there's high value in building architecturally heavy and hard-to-reverse components[11] that tend to produce less visible working software.

For example, we were producing an intranet application slated to have about 300 data entry screens. We could have produced as many screens as possible in the first few sprints to show fast progress and inspire stakeholder confidence. Instead, we persuaded the product owner to let the team build a flexible input field-editing design with high reusability across all screens. This resulted in fewer screens in the early sprint reviews, but it increased the system's maintainability. Toward the end of the project, it allowed screen production at a rate not possible had we not carefully explained to the product owner the early architectural work's value and gained approval to do it.

Architecture Backlog

Like all stakeholders, the architect will want to put functionality into the product more quickly than project velocity will allow. This requires some functionality (in this case architectural) to be placed on the product backlog. As with all use of the product backlog, the work is placed in priority order; and if there isn't enough time or money to get to the work, it might not get done. You could argue that this results in a compromised architecture. Certainly if architectural work receives such a low priority that it never gets done, the architecture will degrade. But proper use of product backlog principles and proper advocacy with the product owner should result in highly valuable architectural work getting done, with less valuable architectural work potentially not happening before the project stops.

To keep a clear focus on architecture and to facilitate architectural scoring, a separate but connected product backlog called the *architecture backlog* should track the architecture work. According to most literature, there's just a single product backlog. In practice, I've found it useful to maintain several physical product backlogs, each focused on their purpose but all collectively serving as the one logical product backlog. The *out-of-scope* backlog clarifies what isn't the goal, the *wish list* backlog lists work that will probably never get done, and so on. Such modularization helps keep the clarity of the main backlog as high as possible without losing a full perspective. So it is with the architecture backlog. It's maintained by the architect, communicated to the product owner and team at the appropriate times and places, and has its items moved to the main backlog on the basis of the product owner's judgment as influenced by the architect. I've found it particularly helpful to give the items a weight and score that provide a grade of the project's architectural quality as it executes. Such scoring provides a clear, measurable mechanism for encouraging the product owner to move stories from the architecture backlog to the main backlog and get them done.

Incremental Enterprise Architecture

The approach to architecture I've advocated so far focuses entirely on project execution on the assumption that the architect's single best opportunity to move system architecture in the right direction comes from guiding the increments of

working software built during sprints. Much of this directional influence focuses on meeting the project's business objectives, but maximum value to the larger organization requires supplementing this guidance with an enterprise architecture (EA) perspective.

Using the four functions we're limiting ourselves to here, I define EA as the process of ensuring that architects

- draw from a uniform hardware and software stack,
- leverage the same design patterns and design pattern language,
- score against the same quality attributes using the same definitions and a uniform scoring scale,
- do each of these on both an intra- and intersystem basis, and
- communicate with each other and their product owners.

The intersystem requirement is based on the observation that system interaction potentially introduces a new layer of design patterns[15] and might shift the overall quality attributes—for example, two systems can scale individually but don't scale when they interact. Most of this definition of EA is independent of agile, but from an agile perspective, the key aspect is the communication and collaboration it requires among the architects and from the architects to the technical teams and product owners.

Communication among the architects is best achieved by having a centralized EA practice and formal EA processes. Senior management must create this practice, ensure that it facilitates and measures communication among the architects, and fund it to the degree they're serious about achieving good EA. Once formed, this practice must establish formal processes and tools, such as an architectural steering committee that publishes uniform architectural scores, a peer review process that checks for issues and provides actionable improvements, and stewardship of a growing body of standards derived from project work. But at all times, two core agile considerations must dominate. First, this is a community of collaborating individuals, not just a process or a collection of artifacts. Second, the power of this process isn't its formal authority but the legitimacy it derives from its architects' expertise and their direct participation in project work.

Communication to the agile teams and product owners is best achieved by physically decentral-

About the Author

James Madison is a senior information architect at a large insurance company and the primary instructor for agile training in the enterprise architecture department. His agile projects include Web, full-client, service-oriented architecture, data warehousing, and projects not traditionally agile such as infrastructure building and platform migration. Madison has a master's degree in computer science from Rensselaer Polytechnic Institute. Contact him at madjim@bigfoot.com.

izing architects and having them incorporate EA concerns at the interaction points. The centralized activities I've advocated so far are an important, but not major, part of the architect's time. An architect should spend as much time as is reasonable physically collocated with the team and product owner to maximize opportunities for direct communication. As the architect advocates aspects of the architecture for the project work, he or she must incorporate EA concerns. For example, when advocating a certain hardware and software stack, base it on not just the project's needs but also the desired EA direction—likewise with design patterns and quality attributes. This is particularly important for intersystem concerns. The architect is uniquely positioned to understand intersystem dynamics that the team and product owner might not, giving the architect a unique responsibility to make hidden problems clear or identify broader opportunities that others might not see. Most of the focus, for everyone including the architect, will likely remain on the business functionality for which the project was funded and on the short-term execution challenges that emerge on any project. But with a reasonably solid vision of the EA's goals and effective incorporation of good architecture in every sprint on every project across time, the EA state should steadily improve.

For example, in 2009 my company won an industry award for "Creating an Agile Business Intelligence Infrastructure."[16] The company funded the project for business reasons in 2008, but the EA community conceived the architecture in 2006—two years before any opportunity existed to deliver it. The problem was that the research community operated on a platform isolated from the main data warehouse, resulting in extreme siloing, high data redundancy, and inadequate operational procedures. At the same time, the main data warehouse used technologies that didn't meet the researchers' needs and had operating norms too restrictive for research work.

On the basis of many years of working with the two departments and understanding their unique cultures and environments, the architects proposed

building a middle ground: a layer that used the research platform's technologies along with the data warehouse's operational procedures and centralized data assets, but adapting each to balance researcher flexibility, system maintainability, and operational efficiency. The EA solution sat on the shelf for two years. Once a project presented the opportunity to move the EA solution forward, the architects leveraged the project, the success was recognized in both the organization and the industry, and the architecture's reusability makes it attractive for future projects.

Agility and architecture aren't at odds. Agile development gives the architect repeated opportunities to work closely with the business and technical teams to continually guide systems in the direction of good architecture. Doing so presents challenges, some inherent in the difficulty of achieving good architecture regardless of methodology, some caused by having to drive to long-term outcomes using a series of short-term events. By simplifying agile methods to a perspective such as the one presented here and being influential at the critical interaction points, a skilled architect can adapt to agile development while staying focused on the core architectural work. This will ensure that both individual systems and their aggregate enterprise behavior meet the needs of the business today, and are technically sustainable for years to come—an architectural value proposition that's independent of delivery methodology. ⑩

References

1. K. Schwaber, *Agile Project Management with Scrum*, Microsoft, 2004.
2. K. Beck and C. Andres, *Extreme Programming Explained: Embrace Change,* 2nd ed., Addison-Wesley Professional, 2004.
3. M. Fowler, *Patterns of Enterprise Application Architecture*, Addison-Wesley Professional, 2002.
4. E. Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, 1995.
5. L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice,* 2nd ed., Addison-Wesley Professional, 2003, pp. 71–98.
6. R. Kazman, M. Klein, and P. Clements, *ATAM: Method for Architecture Evaluation,* tech. report CMU/SEI-2000-TR-004, ESC-TR-2000-004, Software Eng. Inst., Carnegie Mellon Univ., 2000.
7. M. Poppendieck and T. Poppendieck, *Lean Software Development: An Agile Toolkit for Software Development Managers,* Addison-Wesley Professional, 2003, pp. 38–45, 103–111.
8. K. Schwaber and M. Beedle, *Agile Software Development with Scrum,* Prentice Hall, 2002, pp. 23–30.
9. R. Kimball and M. Ross, *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling,"* John Wiley & Sons, 2002, pp. 78–88.
10. V. Subramaniam and A. Hunt, *Practices of an Agile Developer: Working in the Real World,* Pragmatic Bookshelf, 2006, pp. 155–157.
11. M. Fowler, "Who Needs an Architect?" *IEEE Software*, vol. 20, no. 5, 2003, pp. 11–13.
12. M. Ross and R. Kimball, "Differences of Opinion," *Intelligent Enterprise*, 6 Mar. 2004.
13. J. Madison, *Very Large Calculation Systems*, Casualty Actuarial Soc., 2009.
14. J. Shore and S. Warden, *The Art of Agile Development*, O'Reilly, 2008, pp. 214.
15. G. Hohpe and B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*, Addison-Wesley Professional, 2003.
16. "Best Practices in Business Intelligence: Creating an Agile BI Infrastructure," *Computerworld*, 2009; www.biperspectives.com/awards.aspx.