How to Agilely Architect an Agile Architecture

by Stephany Bellomo, Philippe Kruchten, Robert L. Nord, and Ipek Ozkaya

AGILITY AND ARCHITECTURE

The phrase "Agile architecture" evokes two concepts:

1. A system or software architecture that is versatile, easy to evolve, and easy to modify, while resilient enough not to degrade after a few changes.
2. An Agile way to define an architecture, using an iterative lifecycle, allowing the architectural design to tactically evolve over time, as the problem and the constraints are better understood.

The two concepts are not the same: you can have a non-Agile development process that leads to a flexible, adaptable architecture, and vice versa, an Agile process that leads to a rigid and inflexible architecture. One does not imply the other. In the best of worlds, though, we'd like to have an Agile process that leads to a flexible architecture.

The eleventh principle behind the Agile Manifesto[1] — "The best architectures, requirements, and designs emerge from self-organizing teams" — reinforces the belief among some teams that an architecture will gradually emerge out of applying Agile practices, such as biweekly refactorings. This thinking was cemented by mantras such as YAGNI (You Ain't Gonna Need It) and No BDUF (No Big Design Up Front), as well as a belief in deferring decisions to the last responsible moment. Principle 11, however, is neither prescriptive nor testable.[2]

This thinking about the spontaneous emergence of architecture is reinforced by experiences with IT software endeavors that do not require a significant amount of bold new architectural design because (1) the most important design decisions have been made months earlier, (2) they are fixed by current preexisting conditions, or (3) they are the result of a de facto architectural setup in a specific domain. These architectural decisions are already embodied within the choice of frameworks and off-the-shelf software packages. The operating system, servers, programming language, database, middleware, and other choices are predetermined in the vast majority of these software development projects, or the project team has a very narrow range of choices. There is in fact little significant architectural work left to be done.

Architectural design, when it is really needed because of a project's novelty, has an uneasy relationship with traditional Agile practices. Unlike the functionality of a system, design cannot easily be decomposed into small chunks of work, user stories, or "technical stories." Most of the difficult aspects of architectural design are driven by nonfunctional requirements, or quality attributes: security, high availability, fault tolerance, interoperability, scalability, and so on. Other difficulties are driven by quality attributes related to development itself — such as testability, certification, and maintainability — which cannot be parceled up and for which tests are difficult to produce up front. Key architectural choices cannot be easily retrofitted on an existing system by means of simple refactorings. Some of the late decisions may gut out large chunks of the code, and therefore many of the architectural decisions have to be made early, although not all at once.

Many practitioners have grappled with the issue of marrying an Agile approach to designing a solid architecture, such as Cutter Senior Consultant Alistair Cockburn and his "walking skeleton"[3] or Dean Leffingwell and his colleagues' Scaled Agile Framework® (SAFe™).[4] Common thinking nowadays is that architectural design and the gradual building of the system (that is, its user-visible functionality) must go hand in hand, but there are several delicate issues:

- How do we pace ourselves?
- How do we make decisions over time in a way that will lead to a flexible architecture and enable developers to proceed?
- In which order do we pick the quality attribute aspects and address them?

The concept of Agile architecture is not new: evolvability, software evolution, and reengineering of existing systems have been studied and understood for a long time. Indeed, Manny Lehman started this investigation circa 1980.[5] The novel challenge that Agile architecting brings to system evolvability is the practices that allow architecting the system in smaller chunks. Successful teams are those that can take advantage of existing software engineering techniques with slight modifications, in particular those that provide early feedback and learning, such as prototyping.[6]

In this article, we present lessons learned about the characteristics of an Agile architecture that enabled an organization to develop its architecture in an Agile manner and continue to rapidly deliver features when more stringent quality attribute requirements emerged.

To investigate why some teams have fewer delays and disruptions to continuous delivery of features than others, we interviewed project teams from several government and commercial organizations. The examples we present here come from one such project team, which was challenged when an unexpected performance quality attribute requirement surfaced in customer feedback during a user demo. Fortunately, the team was able to react to the emerging requirements without disrupting the project. When we examined the team's practices, we found they were doing Agile architecting that resulted in an Agile architecture.

AGILE ARCHITECTING

Agile architecting is not only the process of allocating architectural work to iterations. There is also a winding route in which development proceeds as the requirements, architecture, and design/implementation are elaborated iteratively and often concurrently. Early understanding of requirements and architecture choices is key to managing large-scale systems and projects; however, linkage between architecture and implementation design choices is also crucial.[7] While requirements originating from the problem space inform architecture and development, explorations originating from architecture and implementation investigations also assist in eliciting and detailing requirements. The essence of Agile architecting is to conduct these activities concurrently with the right balance.

In Figure 1, we illustrate an example (captured during our interviews with the team) of such an Agile architecting process for prototyping with a quality attribute focus.[8] In this example, the team probes for quality concerns during a user demo, discovers an emerging performance requirement (the user expected faster rendering of data-intensive pages than anticipated),

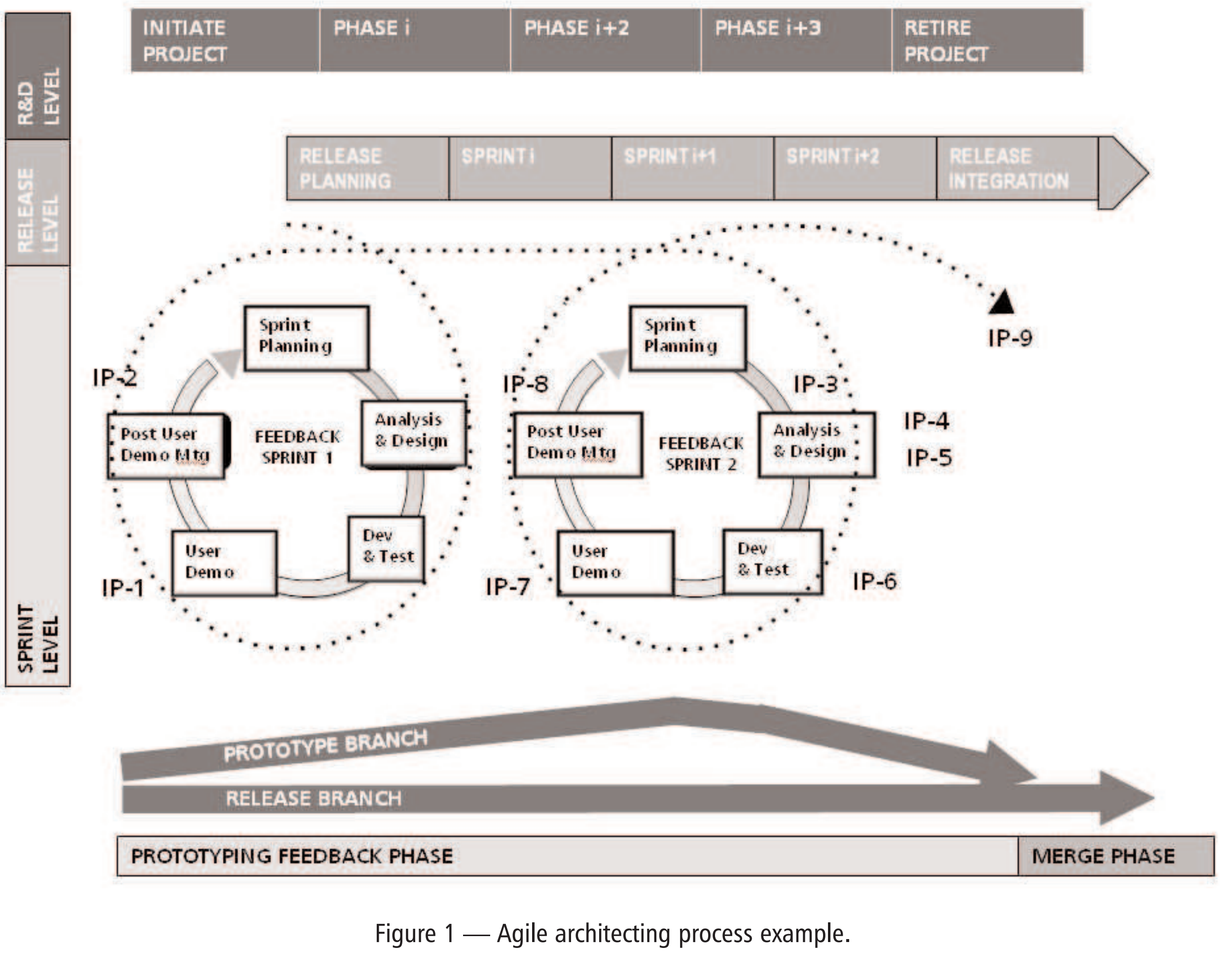

Figure 1 — Agile architecting process example.

analyzes the requirement, architects a solution to the performance problem, implements the solution in a demonstrable prototype, and gets feedback from the user at the next user demo. The user accepts the prototyped implementation, and the team merges the implemented code into the project baseline for a future release.

The team operates in three cycles in which development is iterated through biweekly sprints; the architecture and product team manage the releases through forward-looking architecture-focused investigations; and a separate R&D cycle investigates long-term strategic technology goals. Figure 1 shows the process steps as integration points (IPs) between requirements and architecture or between architecture and design/ implementation:

- IP-1* (Rqmts->Arch): Probe for emerging quality attribute requirements at user demo.
- IP-2 (Rqmts->Arch): Analyze quality attribute requirements.
- IP-3 (Rqmts->Arch): Conduct deeper analysis of emerging quality attribute.
- IP-4 (Rqmts->Arch): Identify architectural approach/patterns.
- IP-5 (Arch->Design/Imp): Elaborate architectural patterns for specific system.
- IP-6 (Arch->Design/Imp): Implement portion of prototyped solution as a spike.
- IP-7 (Rqmts->Arch): Get user feedback on prototype.
- IP-8 (Rqmts->Arch): Analyze feedback on prototype.
- IP-9: Get approval for prototyped changes (merge with release).

**Note that the team probes for emerging requirements during the user demo as part of the requirements elicitation process, which explains why IP-1 starts there.*

Two particularly important practices that helped the team succeed are prototyping prior to the target sprint and prototyping in a separate environment. This continuous architecture exploration allowed the team to focus on making the architecture Agile enough to support upcoming needs, which helped strike a balance between too much up-front architecture and not enough.

The example shown in Figure 1 illustrates the prototyping process the team uses for a relatively small architectural change. The team handles these smaller changes at the sprint/release planning level. For larger-scale infrastructure improvements targeted at future phases, the team explained that they often create an R&D prototype to begin reasoning about foundational architecture that may be needed to support emerging functional and quality attribute requirements (e.g., clustering infrastructure for the scalability quality attribute requirement). The team creates such prototypes in an entirely separate lab environment, so there is no risk to the development environment.

AGILE ARCHITECTURE

We found that the team has an Agile architecture that they described as a "flexible architecture." This allows them to explore technical options rapidly with minimal ripple effect. The architecture can be understood in terms of patterns and tactics that influence the time and cost to implement, test, and deploy changes.[9]

The overarching pattern the team uses to enhance modifiability and control the time and cost of change is the *layered pattern*. Other supporting patterns and tactics to separate interfaces, restrict dependencies, and separate concerns are employed as well. The team described their architecture choices as follows:

- **Layered architecture, client-server architecture, and separation of the presentation from business/data layers.** Over time, the architecture and vision have been implemented as various layers, and components can be replaced wholesale without significant disruption elsewhere. The layered architecture supports adding new client interfaces and replacing presentation components with minimal ripple effect. For example, the presentation layer was modified to add a new Web client to an existing smart client implementation with no effect on the underlying architecture and no down time in operations to introduce the new interface. The smart client was later replaced with .NET C# WPF with no impact to the back end. All three presentation technologies existed simultaneously using the same back end.
- **Service orientation with separate interfaces and restricted dependencies.** The data layer was built behind an API service layer to insulate the client from changes in the data layer. These tactics enable the team to fence off third-party dependencies by creating a metadata definition layer (in C#). They also support replacing the underlying database and access control without impact to the service-oriented data layer, thereby allowing the presentation layer to continue in production without modification.

- **Publish-subscribe pattern.** This pattern is used to monitor the data layer schema for changes since the database is owned and managed by a third party.

The team uses the following patterns and tactics to improve scalability:

- **Clustered architecture with load balancing and replicated copies.** The cluster-based architecture allows rapid scaling, whether increasing for mission need or scaling back due to budgetary constraints. It is possible to scale up by increasing the size of the virtual servers or to scale out by increasing the number of clusters, the number of analysts using the App-V client, or the number of Citrix servers in the cluster.
- **Encapsulation of algorithms.** Computationally intensive algorithms, which have a significant impact on system performance, are encapsulated to allow them to evolve or be replaced.
- **Data caching.** A local copy of a subset of data is kept to optimize performance of data retrieval and data processing for frequently accessed data.

The team also focuses on flexibility in deployment and controlling the cost and time for testing by using several patterns and tactics:

- **Virtualization, layering both the infrastructure and the application.** By adopting virtualization at a very early stage, the system may be hosted in any data center with the appropriate network connections. For example, infrastructure (VMware) and the application (App-V) are virtualized. This enabled the system to be consolidated from deployment in five regional centers to two, allowing the customer to reduce costs.
- **Standardized and configurable architecture: parameterization and static and dynamic binding.** Each cluster has a standard configuration and can be customized. This standardization allows installations to be accomplished at will; for example, the team can allocate new servers (in the cloud), install the software, and replicate joins to scale the system to meet new needs. Additional processors and memory can be added to a virtual machine to increase capacity. Similarly, clusters can be removed at will to meet resource constraints.
- **Executable, interface-driven code structure.** Executable code structure enables automated testing of business layer functionality through interfaces.

A solid understanding of the overall architectural structure helps the team to respond to stakeholder feedback or learning from spikes (that is, prototyping activities that are timeboxed[10]), because they have the architectural knowledge to rapidly analyze the impact of changes and conduct architecture tradeoff analysis.

INTEGRATING PRACTICES WITHIN INCREMENTAL DEVELOPMENT ENVIRONMENTS

Understanding the *desired state* of development and delivery helps tie together Agile architecting and Agile architecture. A desired software development state is one that enables Agile teams to quickly deliver releases that are valuable for stakeholders.[11] The teams themselves typically define the desired state. It is their vision of the ideal development infrastructure that they would like to work with. When product development starts, the desired state does not necessarily exist. Setting up the initial architecture helps push the team and the delivery into the desired state and enables Agile architecting that results in an Agile architecture.

In this example, the team needs both Agile architecting and architecting for agility to stay within the desired state. Agile architecting provides the team a regular cadence for periodically considering whether they are already out of bounds or about to get out of bounds. The regular cadence of Agile architecting also helps the team manage architecture exploration and balance competing concerns of too much or too little architecting.

Agilely architecting an Agile architecture, then, minimally has four key requirements:

1. Focusing on key quality attributes and incorporating these into technical explorations within prototyping and spikes
2. Understanding that a successful product is a combination of not only customer-visible features but also the underlying infrastructure that enables those; hence, architectural requirements are incorporated into sprint planning and demos
3. Recognizing that an Agile architecture that enables ease of maintainability and evolvability is the result of ongoing, explicit attention given to the architecture, not a natural byproduct of an Agile — or any — software development process
4. Continuously managing and synchronizing dependencies between functional and architectural requirements and ensuring that the architectural foundation is put in place in a just-in-time manner

At a high level, the process for Agilely developing an Agile architecture can be seen as a zipper, as shown in

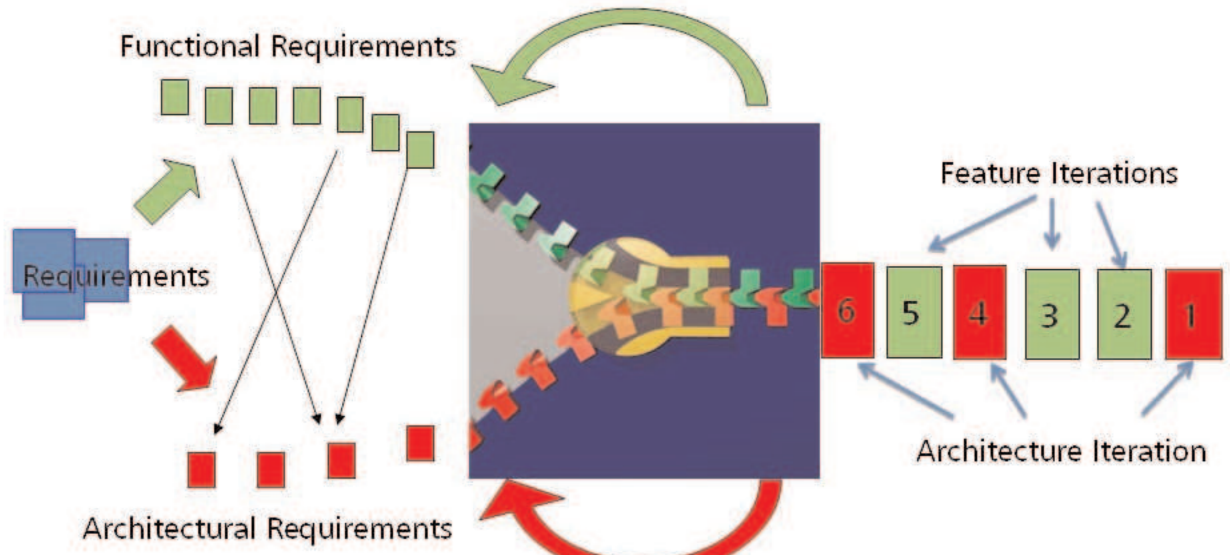

Figure 2 — The zipper model.

Figure 2. From the requirements, as they evolve, the designers extract and separate functional requirements (mostly features seen by the end users) and architectural requirements (mostly derived from key quality attributes), which are already expressed or anticipated. Dependencies between these two kinds of requirements must be managed to ensure that necessary elements of the architecture are present (or at least "stubbed") in upcoming iterations with functional requirements that depend on them. This skeletal foundation must be woven into early iterations of architectural and functional increments.

This approach will facilitate a deliberate (not accidental) emergence of an architecture, constantly validated by the functionality developed on top of it. The development of the architecture occurs over several iterations, without being stopped, blocked, or slowed down by developers claiming "YAGNI" or "No BDUF." However, as when a zipper gets out of alignment, causing it to get stuck, teams that do not pay close attention to evolving dependencies can get caught off guard without the architectural foundation they need to support emerging requirements. Agility and architecture do support each other very well.

ACKNOWLEDGMENTS

We would like to acknowledge Sphere of Influence for their technical contributions to this article. This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

ENDNOTES

[1]"Principles Behind the Agile Manifesto" (http://agilemanifesto.org/principles.html).

[2]Séguin, Normand, Guy Tremblay, and Houda Bagane. "Agile Principles as Software Engineering Principles: An Analysis." *Lecture Notes in Business Information Processing*, Vol. 111, edited by Claes Wohlin. Springer, 2012.

[3]Cockburn, Alistair. "Walking Skeleton" (http://alistair.cockburn.us/Walking+skeleton).

[4]Scaled Agile Framework® (http://scaledAgileframework.com).

[5]Lehman, Meir M. "Programs, Lifecycles, and Laws of Software Evolution." *Proceedings of the IEEE (Special Issue on Software Engineering)*, Vol. 68, No. 9, September 1980.

[6]Ozkaya, Ipek, Robert L. Nord, Stephany Bellomo, and Heidi Brayer. "Beyond Scrum + XP: Agile Architecture Practice." *Cutter IT Journal*, Vol. 26, No. 6, 2013.

[7]Nuseibeh, Bashar. "Weaving the Software Development Process Between Requirements and Architectures." *IEEE Computer*, Vol. 34, No. 3, March 2001.

[8]Bellomo, Stephany, Robert L. Nord, and Ipek Ozkaya. "Elaboration on an Integrated Architecture and Requirement Practice: Prototyping with Quality Attribute Focus." *Proceedings of the Second International Workshop on the Twin Peaks of Requirements and Architecture*. IEEE, 2013.

[9]Bass, Len, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. 3rd edition. Addison-Wesley Professional, 2012.

[10]Leffingwell, Dean. *Agile Software Requirements: Lean Requirements Practices for Teams, Programs, and the Enterprise*. Addison-Wesley Professional, 2011.

[11]Bachmann, Felix, Robert L. Nord, and Ipek Ozkaya. "Architectural Tactics to Support Rapid and Agile Stability." *CrossTalk*, May/June 2012.

Stephany Bellomo is a senior member of the technical staff at the Carnegie Mellon University (CMU) Software Engineering Institute (SEI). Ms. Bellomo is a member of the Architecture Practices group and an active member of the Value-Driven Incremental Development research team. She teaches SEI courses in Service-Oriented Architecture Migration of Legacy Components and Software Architecture Principles and Practice. Ms. Bellomo has over 15 years' experience in the software field. Prior to joining the SEI, she worked as a software engineer for several organizations, including Lockheed Martin, Intuit, and VeriSign Network Solutions. She served as tutorial chair for SEI's SATURN Conference *and is currently a member of the organizing committee for the* International Workshop on Release Engineering 2014 *(hosted by Google). Ms. Bellomo received an MS in software engineering from the George Mason University. She can be reached at sbellomo@sei.cmu.edu.*

Philippe Kruchten is a professor of software engineering at the University of British Columbia (UBC), in Vancouver, Canada, where he holds an NSERC Chair in Design Engineering. Dr. Kruchten joined UBC in 2004 after a 30-plus-year career in industry, where he worked in large software-intensive systems design in the domains of telecommunications, defense, aerospace, and transportation. Some of his experience is embodied in the Rational Unified Process, whose development he directed from 1995 to 2003. His current research interests reside mostly with software architecture, in particular architectural decisions and the decision process, as well as software engineering processes, especially the application of Agile processes in large and globally distributed teams. Dr. Kruchten teaches courses in entrepreneurship, software project management, and design. He is a senior member of IEEE Computer Society; an IEEE Certified Software Development Professional; a member of ACM, INCOSE, and CEEA; the founder of Agile Vancouver; and a professional engineer in British Columbia. Dr. Kruchten has a diploma in mechanical engineering from Ecole Centrale de Lyon and a doctorate degree in information systems from Ecole Nationale Supérieure des Télécommunications in Paris. He can be reached at pbk@ece.ubc.ca.

Robert L. Nord is a senior member of the technical staff at the SEI. Dr. Nord is engaged in activities focusing on Agile and architecting at scale and works to develop and communicate effective methods and practices for software architecture. His collaboration with Philippe Kruchten and Ipek Ozkaya is helping shape the research agenda on technical debt. He is coauthor of the practitioner-oriented books Applied Software Architecture *and* Documenting Software Architectures: Views and Beyond *and lectures on architecture-centric approaches. Dr. Nord is a member of the steering committee of the* WICSA Conference *series, in addition to organizing events at software engineering, Agile, and architecture venues. He earned a PhD in computer science from CMU and is a distinguished member of the ACM. He can be reached at rn@sei.cmu.edu.*

Ipek Ozkaya is a senior member of the technical staff at the SEI. With her team, Dr. Ozkaya works to help organizations improve their software development efficiency and system evolution. Her work focuses on software architecture practices, software economics, and requirements management, and her latest publications include articles on Agile architecting, dependency management, and architectural technical debt. Dr. Ozkaya also chairs the advisory board of IEEE Software *and serves as an adjunct faculty member for the Master of Software Engineering Program at CMU. She also organizes different events (tutorials, workshops, and sessions) and is an invited speaker at software engineering, Agile, and architecture venues (e.g.,* ICSE, OOPSLA, SATURN, *and* WICSA*). She holds a doctorate from CMU. She can be reached at ozkaya@sei.cmu.edu.*