Agile in Distress: Architecture to the Rescue

Robert L. Nord[1], Ipek Ozkaya[1], and Philippe Kruchten[2]

[1] Carnegie Mellon Software Engineering Institute, Pittsburgh, PA, USA
{rn,ozkaya}@sei.cmu.edu
[2] Electrical & Computer Engineering, University of British Columbia, Vancouver, Canada
pbk@ece.ubc.ca

Abstract. For large-scale software-development endeavors, agility is enabled by architecture, and vice versa. The iterative, risk-driven life cycle inherent in agile approaches allows developers to focus early on key architectural decisions, spread these decisions over time, and validate architectural solutions early. Conversely, an early focus on architecture allows a large agile project to define an implementation structure that drives an organization into small teams, some focusing on common elements and their key relationships and some working more autonomously on features. Architects in agile software development typically work on three distinct but interdependent structures: architecture of the system, the structure of the development organization, and the production infrastructure. Architectural work supports the implementation of high-priority business features without risking excessive redesign later or requiring heavy coordination between teams. Architectural *tactics* provide a framework for identifying key concerns and guide the alignment of these three structures throughout the development life cycle.

Keywords: agile, architecture, organizational structure, production infrastructure, large-scale agile software development, software engineering, project management.

1 Introduction

Agile software-development approaches have provided notable improvements over more rigid, phased, document-intensive approaches. This should not be surprising: many of the practices that the Agile Manifesto encourages had existed for a while, but first the mindset of software design and development practitioners had to shift. These approaches now emphasize trust, face-to-face communication, and less formal artifacts coupled with new technologies for computer-supported communication and development environments. Agile approaches work well for projects in a "sweet spot" with certain enabling characteristics: small teams of 5–12, preferably collocated; a stable underlying architecture; frequent deliveries; and low to medium criticality of the system. But the question "how do we scale 'agile' to larger, bolder software-development endeavors?" is still repeatedly asked.

In this paper, we define *large scale* by scope of the system, team size, and project duration. At a large scale, the scope of the system touches several domains and has some combination of interoperability, security, and performance concerns. The size of a development team is more than 18 people, partitioned into a few teams, and likely to be geographically distributed. And the duration of the development typically extends beyond a year.

There are many possible answers to the questions "how do we scale agile up?" and "how do we use it outside of its sweet spot?" They often take the form of modifying an agile practice to make it work "at scale." The typical example is the daily standup meeting, or *Scrum*, scaled up to a *Scrum of Scrums*. But this may be a solution to a different problem. The real problem is how to be agile at the organization level, not simply how to scale individual practices (even if the latter may support the former).

In this paper, we demonstrate that an early and continuous focus on the *architecture* of the system enables scaling up agile development and minimizing unanticipated roadblocks. If we define *agility* as the ability of an organization to rapidly react to change in its environment [1], [2], then we can restate the problems as follows:

- Can the architecture of the product support multiple waves of enhancements, to accommodate a constant flow of new needs?
- Can the architecture evolve continuously to support enhancements?
- Does the architecture allow teams to organize the work so that they feel as if they were in the sweet spot and allow them to take advantage of agile practices?
- Can the organization avoid the extra work generated by repetitive handover from a development team to an operations group?

2 Why Scale Necessitates Architecture

Architecture is the high-level structure of a software system, the discipline of creating such a high-level structure, and the documentation of this structure [3]. The architecture of a software system is a metaphor, analogous to the architecture of a building [4]. Agile teams sometimes fear architecture as a remnant of some ugly past, decry it as "big design upfront" (BDUF), and naïvely hope that a suitable architecture will gradually emerge out of weekly refactorings. While "refactoring has emerged as an important software engineering technique, it is not a replacement for sound architectural design; if an architecture is decent you can improve it, but refactored junk is still junk" [5]. We know that at-scale development needs a healthy, proactive, and early focus on both system architecture and software architecture.

Architecture provides a way to partition work around large chunks of software development, guiding the organization into teams. This often takes the form of one or more "infrastructure" teams supporting one or more "feature" teams. Conflicts in software development are reduced when there is an overall socio-technical congruence between the structure of the system and the structure of the teams. This allows the creation of islands of stability in which teams can operate in a mode that is closer

to the agile sweet spot, and possibly at a faster iteration rhythm. But architecture also provides other benefits to a large, distributed project:

- a common vocabulary and a common culture to speak about the system and how it functions. This was the intent of the "metaphor" practice of the original XP.
- a systematic way to control dependencies—of code, data, timing, and requirements—which tend to grow uncontrolled in large projects.
- a way to keep technical debt in check, by identifying and gradually reducing technical debt at the structural or architectural level, which is the second type of debt in McConnell's taxonomy [6].
- a guide for release planning and configuration management.

When projects scale up on all three dimensions of scope, team size, and duration, software developers need tools to organize the work, make the right decisions, communicate these decisions, implement and validate them, and define guidelines and processes applicable across the project. These tools do not exist in the traditional toolkit of Scrum, XP, and lean, but they can be found in the architects' toolkit.

When working at scale, the agile community begins in small ways to acknowledge the need for architecture, and even sometimes for an architect (or "architecture owner," as a counterpart to the "product owner"), as, for example, in Cockburn's Walking Skeleton [7] and the architectural runway of the Scaled Agile Framework [8]. We have seen increasing evidence in practice where successful teams tailor architecture with agile approaches [9], [10], [11].

In practice, architects in agile software development typically work on three distinct but interdependent structures (Fig.1):

- The *Architecture* (A) of the system under design, development, or refinement, what we have called the traditional system or software architecture.
- The *Structure* (S) of the organization: teams, partners, subcontractors, and others.
- The *Production infrastructure* (P) used to develop and deploy the system, the last activity being especially important in contexts where the development and operations are combined and the system is deployed more or less continuously.

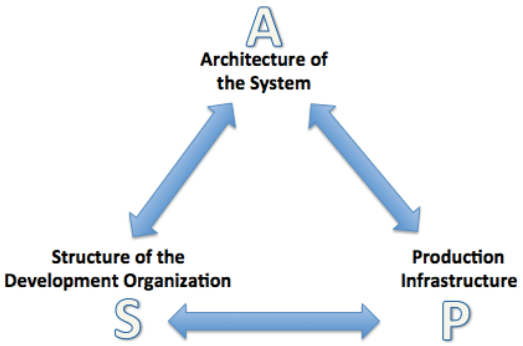

Fig. 1. System architecture (A), organizational structure (S), and production infrastructure (P)

These three structures must be kept aligned over time to support agility. In this paper, we examine the alignment of these structures from the perspective of A and the role of the architect in an agile software-development organization. The relationship of A to S is known as *socio-technical congruence* [12] and has been extensively studied, especially in the context of global, distributed software development. It is very pertinent at the level of the static architectural structure (development view), where a development team wants to avoid conflicts of access to the code between teams and between individuals, while having clear ownership or responsibility over large chunks of code. When A is lagging, we face a situation of technical debt [13]; when S is lagging, we have a phenomenon called "social debt," akin to technical debt, which slows down development [14].

The alignment of A with P is seeing renewed interest with increased focus on continuous integration and deployment and the concept of "DevOps" [15]: combining the development organization with the operations organization, and having the tools in place to ensure continuous delivery or deployment, even in the case of very large online, mission-critical systems (e.g., Netflix, Facebook, Amazon). When P is lagging, we witness a case of "infrastructure debt" as described by Shafer [16], which is another source of friction in software development.

A, S, and P must be "refactored" regularly to be kept in sync so that they can keep supporting each other. Too much early design in any of the three will potentially result in excessive delays, which will increase friction (by increased debt), reduce quality, and lead to overall product delivery delays.

3 How Architecture Benefits from Agility

Given that software designers and developers increasingly recognize the importance of architecture in supporting large-scale agile development, the challenge is no longer about whether architecture is needed but about *when* is it needed, how often, and when the misalignment of A with S and P should trigger a large or small refactoring of the whole A-S-P triad. Designing an architecture in one large increment upfront could delay feedback on the requirements and technical risks of the system.

Architectural design benefits from agility primarily through shorter iterations that produce smaller increments and provide earlier feedback: architectural design and the gradual building of the system go hand in hand. As the stakeholder needs evolve, the designers extract functional and architectural requirements. Dependencies between these two kinds of requirements must be managed to ensure that necessary elements of the architecture are present (or "stubbed") in upcoming iterations. This skeletal foundation must be woven into early iterations of architectural and functional increments. This approach facilitates a deliberate emergence of an architecture over several iterations, constantly validated by the functionality developed on top of it.

This process raises however several practical challenges:

- How do we pace ourselves? What is an appropriate increment? In which order do we work on the functionality and architecture to produce an increment of value to the user while managing costs of rework and delay?

- How do we use iterations to also refine and evolve both S, the structure of the development organization, and P, the production infrastructure.

We've described this approach using the "zipper" metaphor [17]; see Fig. 2. As the requirements are being developed and refined, the architect identifies and extracts architecturally significant requirements (in red); more feature oriented, or functionally oriented, requirements (in green); as well as dependencies between them: *to start building story card X, we need architectural support Y that is at least prototyped (e.g., an API and a stub)*. Small iterations are used to design, build, and test a few of the architectural elements and the features that depend on them. By starting with the most critical or challenging requirements, we force the architects to think about the more fundamental aspects early. Whatever they design in the architecture (A) is validated not by some abstract test but by a product embryo of actual functionality.

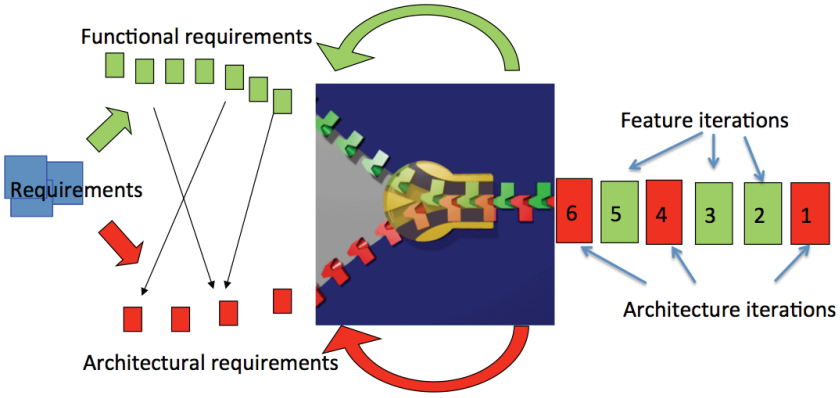

Fig. 2. The zipper model [17]

Some functional and architectural requirements can be in the same iteration, or we can alternate some iterations focusing on architecture and some focusing on more user-facing functionality. Architecture-only iterations may be necessary if some significant refactoring of the emerging architecture is needed, disrupting work on new features to rework existing code.

Many agile practices will therefore contribute to helping architects do a better job of producing more value by reducing waste: frequent communication among the architect, the teams implementing the architecture, and the teams working on features; early realistic testing of certain architectural aspects; reflection and rapid feedback on the suitability or performance of the architecture, allowing for improvements or change of approach; and reduced amount of documentation produced and handed over (compared with an early and more massive architectural undertaking).

This iterative approach, with a clear focus on producing production-quality code early, will also push the architect to pay attention to the allocation of the architecture to the organization and production infrastructure. As the team grows and its structure evolves, the need to improve direct, rapid communication between team members will

lead to frequent adjustments of the team structure (S), such as changing the composition of the teams, redefining responsibilities, re-arranging the office layout, adopting "information radiators" or web-based communication tools, and other modifications. This is self-organization in action.

The emphasis on executable code at each iteration will drive the team to experiment early with creating a viable production and delivery environment (P) and evolve it, working early with the people in charge of operations.

The zipper model is far from the antipattern of BDUF and its associated problems:

1. Architecture? – Done.
2. Development of the function can now start.
3. Oops, the architecture does not support all these functions? – It's a bit late to tell us; the architects have moved on to the next project.

Agile practices and principles support architecture in a tight, integrated fashion, which in turn enables scaled agile development.

4 Architectural Tactics to Support Scaled Agile Development: Exploring the Alignment of A and S

The work assignment allocation view captures the alignment of the architecture and the structure of the development organization [3]. It describes the mapping between the software's modules and the people, team, or organizational work units tasked with developing those modules. The work assignment view helps with planning and managing team resource allocations, assigning responsibilities for builds, and explaining the structure of the project.

Architectural tactics enable a simultaneous focus on architecture and agile development by aligning feature-based development and system decomposition to minimize coupling between teams [18]. The tactics we explore for improving the alignment of the architecture of the system and the development of the organization include vertical and horizontal decomposition of the architecture to enable alignment of the teams accordingly as well as matrix augmented-role team structures.

4.1. Vertical and Horizontal System Decomposition

The system decomposition tactic allows assigning responsibilities to the development teams according to the stage of the development effort and the need to focus on features or infrastructure.

The work assignment view gives each team its charter. A common charter for agile teams is to give them responsibility for every piece of implementation for developing a feature, so they do not have to wait until someone else has finished other work. We call this *vertical decomposition* because every component of the system required for realizing the feature is implemented only to the degree required by the team.

An alternative charter is to give teams responsibilities based on system infrastructure. We call this *horizontal decomposition*, an approach in which an agile team bases

system decomposition on the architectural needs of the system, focusing on a framework of common services and variability mechanisms. To develop a feature, the team implements only the logic of that feature using the frameworks. The frameworks and common services have already taken care of the logic of integrating the new pieces of code into the system. This type of architecture minimizes the dependencies between different feature implementations so that different teams can implement features without coordination.

The larger the system to be developed and the more agile development teams there will be to develop it, the more the underlying architecture has to support independent development teams. A development effort with only one collocated team may not need explicit architecture design and documentation tasks, and the amount of rework when letting the architecture emerge might be acceptable. With more teams involved, the coordination required between teams starts hindering progress. Each time teams have to coordinate with each other, they may have to wait, which increases the risk of not finishing a task as planned. According to Conway's Law [19], minimizing coordination between teams requires an architecture that is designed to have loosely coupled components so that the team structure can be aligned along those components.

Another factor that makes architecture tasks more important in a large-scale development is the discovery and creation of reusable code, such as common services. If agile development teams are organized to work according to user-visible features (vertically), then the potential of code pieces being reused across features is difficult to evaluate. In this situation, every team is rapidly developing their features, but altogether the teams could have been more efficient if someone would have spent the time discovering and creating a common service (horizontally).

The goal of creating a feature-based vertical decomposition for alignment between architecture and team structure is to decouple teams and architecture to ensure parallel progress where teams are organized in a Scrum of Scrums. Defining the appropriate architecture is key to the success of large-scale software-development projects where there is the need to manage multiple agile teams concurrently over many years. Feature-based vertical decomposition is the preferred approach for assigning tasks to teams. This approach requires minimizing the number of technical and social dependencies to achieve appropriate productivity of the agile development teams.

Two generic examples of such architectures are shown in Fig. 3. Here we have an architecture consisting of three layers. Those three layers contain the common services. Every layer also has either a framework or a plug-in interface defined that implements the control logic of that layer. To develop a feature, only the logic of that feature has to be implemented in each layer using the frameworks or plug-in interfaces. This focuses the development on only what is needed for the feature implementation. All the logic of how to integrate the new pieces of code into the system, such as using the intra-layer communication protocols, is already taken care of by the frameworks and the common services. This type of architecture also minimizes the dependencies between different feature implementations so that different teams can implement features without coordination.

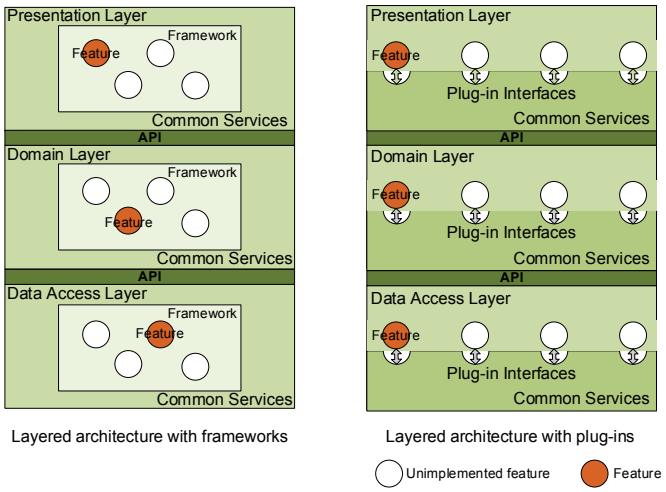

Fig. 3. Layered architecture supporting feature-based development

Horizontal decomposition of the system and alignment of the teams to the architecture accordingly is most useful when the skeleton of the system is being developed. As teams create a platform containing commonly used services and development environments, either as frameworks or platform plug-ins to enable rapid feature-based development, it is best to focus their effort on the development of those layers rather than on functionality that will cross over multiple aspects of the system. Another key activity that requires horizontal decomposition is when the stable interfaces between key system elements are being defined. When the commonly used services and development environment are sufficiently in place, then the teams can change their focus to features that span the system.

In every agile project that we analyzed [10] [18] [20], we observed a strong desire to achieve vertical decomposition and team alignment. But especially in the beginning of a project, there is also a strong need for horizontal decomposition and alignment to ensure the teams build components that support later feature development. Horizontal alignment is mostly seen as a temporary phase to achieve the "desired state" of vertical alignment. The Eclipse plug-in framework architecture can be viewed as an example of this balance between horizontal and vertical alignment of teams to the architecture. The existing architecture framework, created over multiple releases by the internal team, enables external organizations and teams to develop features on the framework. The existence of such an infrastructure allows for rapid development. The goal of agile teams that need to operate at scale should be to establish such a supportive infrastructure, which evolves over time yet still supports rapid development.

4.2. Matrix and Augmented Team Structures

The matrix teams tactic allows introducing specialized roles, such as the architect, seamlessly to the agile development effort.

Coordination and *congruence* tactics provide the context for achieving successful matrix and augmented team structures. In its simplest instantiation, a Scrum development environment consists of a single co-located Scrum team with the skills, authority, and knowledge required to specify requirements, architect, design, code, and test the system. As systems grow in size and complexity, however, the single Scrum team model may no longer meet development demands. Information about complexity and uncertainty can supplement the work assignment view (to produce what Herbsleb calls the *coordination view* [3] [21]) to provide more detail about which teams need how much communication, collocation, or both, and how that communication will influence which strategy to use to affect project structure. Achieving *congruence*, then, is matching coordination requirements and coordination capabilities.

A number of different strategies can be used to scale up the overall development organization while maintaining an agile Scrum-based development approach. One approach is replication, essentially creating multiple Scrum teams with the same structure and responsibilities, sufficient to accomplish the required scope of work. This approach works only to some extent, as typically scale issues are not resolved simply by a Scrum of Scrums, in other words, by more of the same scaling. Successful scaling and alignment of the development organization with the system is mostly achieved by a hybrid approach. The hybrid approach involves Scrum team replication but also changes the nature of the Scrum teams in a number of ways. For example, teams aligned horizontally could use the Scrum of Scrums to coordinate vertical issues; later, as the teams move alignment vertically, the role of the Scrum of Scrum changes to coordinate horizontal issues. Another example is to supplement Scrum teams with traditional function-oriented teams, such as using an Integration and Test team to merge and validate code across multiple Scrum teams or dynamically allocating teams depending on the nature of high-priority tasks. (In purist Scrum circles, the hybrid approach would most likely be labeled an example of "ScrumBut.") At scale, the tasks assigned to teams also need to focus on the alignment. In addition, at scale we often observe the breaking of the self-organization of the teams and a balance between a hierarchical ownership structure and small teams in which roles might be more fluid.

5 Architectural Tactics to Support Scaled Agile Development: Exploring the Alignment of A and P

The install allocation view captures the alignment of the architecture and the production infrastructure [3]. It describes the mapping between the software's components and structures in the file system of the production environment. Understanding the organization of the files and folders of the installed software can help developers, deployers, and operators create build-and-deploy procedures, update and configure files of multiple installed versions of the same system, and design and implement an "automatic updates" feature. With the increasing need to focus on continuous integration and multiple deployment contexts and the growing DevOps movement, this view is becoming more dominant. The alignment of the architecture and the production infrastructure becomes critical to articulate.

The tactics that improve the alignment of architecture and production infrastructure are those that extend the concept of the runway beyond the skeletal architecture stubs to include the tooling infrastructure as well as those that include automated deployment and integration support. The more stable the supporting architecture and infrastructure (platform, frameworks, tools), the more teams can be aligned vertically. The less stable the infrastructure at the onset, the more team members have the responsibility to create parts of that architecture and production infrastructure (necessitating a focus on horizontal decomposition of the system).

5.1 Architecture and Infrastructure Runway

The runway-building tactic applies when there is a need for an architecture and infrastructure sufficient to allow incorporation of near-term needs without potentially introducing delays or extra work. One way to manage the alignment of A and P to enhance agility is to reassess the meaning of the architecture runway. Dean Leffingwell [8] describes his concept of architecture runway as follows: "Architectural runway is the answer to a *big* question: *What technology initiatives need to be underway now so that we can reliably deliver a new class of features in the next year or so?*" As such, establishing the runway is often interpreted as the first iteration of the architecture, selecting the frameworks, packages, and so on. Leffingwell and colleagues also make the statement that the bigger the system, the longer the runway.

Leffingwell, Martens, and Zamora [22] explain the role of intentional architecture as one of the key factors to successfully scaling agile. Building and maintaining architectural runway puts in place a system infrastructure sufficient to allow incorporation of near-term product backlog without potentially destabilizing refactoring.

For systems with a smaller scope (and a smaller team size), a shorter runway—that is, architectural infrastructure to support the present iteration or release cycle—may be all that is needed. Especially in the face of uncertain requirements for technology or features, it may be more efficient for the team to try something out, get feedback, and refactor as needed, rather than to invest more time up front in trying to discern requirements that are in flux.

For systems with increasing scope (and larger teams), a longer runway is needed. Building and re-architecting infrastructure takes longer than a single iteration or release cycle. Delivering planned functionality is more predictable when the infrastructure for the new features is already in place. This requires looking ahead in the planning process and investing in architecture by including infrastructure work in the present iteration that will support future features the customer needs.

However, the meaning of runway must expand to encompass the production infrastructure as well. Often such tasks are covered under the planning phase. Articulating the production environment requires defining the alignment with the architecture to be among the main tasks of the runway construction.

Aligning the teams horizontally and focusing on horizontal decomposition of the system are good practices during the early stages of a project as the architectural runway is created, while vertical alignment works well during feature development. Between those two states, we find matrix structures in which the teams are either hori-

zontally or vertically aligned while some members within those teams have opposite responsibilities. Fig. 4 shows an example of this. Here three teams are horizontally aligned to the layers of the architecture to fulfill their primary responsibility to build infrastructure. However, some team members from each team have the responsibility to develop features, and they coordinate with each other in the Scrum of Scrums.

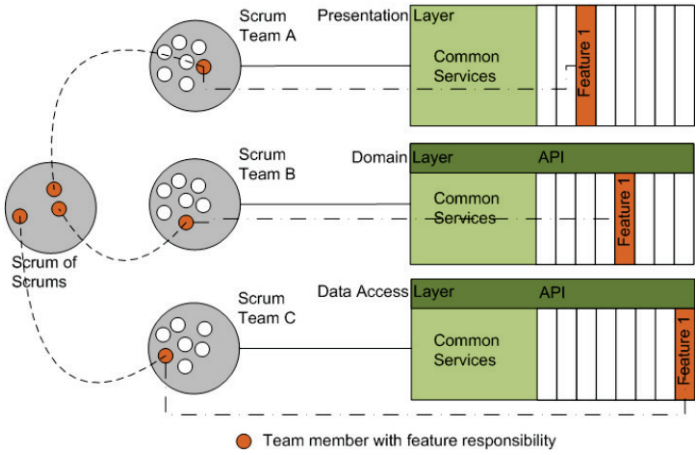

Fig. 4. Progressing architecture and feature development in parallel

5.2 Deployability Tactics

Deployability tactics are those that will make the tooling and deployment environment and alignment run smoothly and at ease. The most relevant tactics include parameterization, self-monitoring, and self-initiating version-update support. While these tactics are also relevant in building the system architecture, they become more significant when managing the alignment of the architecture with the production environment and supporting large-scale operations [23].

Parameterization focuses on environmental variables relevant to the production infrastructure such as databases and server names. This allows deferring binding time and changing aspects of the build and production environment without having to change the build.

Self-monitoring allows for monitoring the system performance and faults as it runs and when it gets out of sync. Both the production infrastructure and the architecture of the system can take advantage of load balancing, logging, and redundancy tactics to realign the allocation and improve system behavior.

Self-initiated version update allows running scripts that update the relevant versions of the software in production. This becomes an issue particularly at scale and when continuous integration and deployment is a goal. The clients and the main applications may get out of sync as well as the supporting tooling environment.

All of these tactics require relevant architecting to influence the allocation relationship between architecture and production infrastructure and to check that the alignment is still in sync.

6 Using the Tactics in Concert to Achieve A-S-P Alignment

In this section, we explore a subset of tactics that can help keep the architecture of the system (A), the structure of the organization (S), and the production infrastructure (P) aligned to achieve agility at scale. Fig. 5 summarizes the tactics that we explored. We did not include tactics related to S-P because we positioned this paper from the perspective of the architecture (A). A complete picture would necessitate exploring alignment tactics for S-P as well as for A-S and A-P.

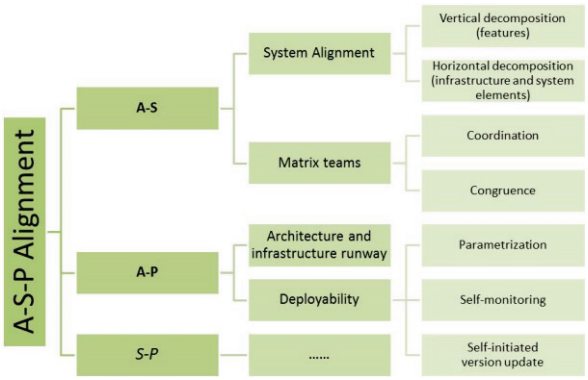

Fig. 5. Summary of the A-S-P alignment tactics

Different phases in a system's life cycle require different tactics. An example walkthrough might look like the following:

At the start of a project, it makes sense to organize the teams horizontally. Most of the team's responsibilities focus on making the supporting infrastructure stable enough for feature development to start. This includes activities related to building the architecture elements (A), understanding the key quality attributes, and establishing the build and deployment infrastructure (P), hence building the architecture runway. Team members create a rough sketch of the architecture, make technology decisions, establish the tool environment, and select relevant deployability tactics. Typically, teams use a small subset of basic features to guide the creation of the development infrastructure, but they may not implement those features during this phase.

As soon as the most important interfaces are defined, some team members start developing features. At this point, a matrix organization is established, focusing on coordination requirements and congruence needs. Most team members still have component-oriented responsibilities; therefore, the teams are still horizontally organized. Now, however, some team members start implementing features using the development infrastructure built so far. For example, in a Scrum of Scrums the team members assigned to implement features coordinate with each other to ensure on-time delivery of the features. This helps stabilize the interfaces and provides the first sketches for implementation frameworks that will be helpful for feature development.

As the interfaces become more stable, most of the teams switch to vertical (feature-oriented) development. Some team members still have horizontal responsibilities because the development of common services as well as framework and interface enhancements is performed continuously.

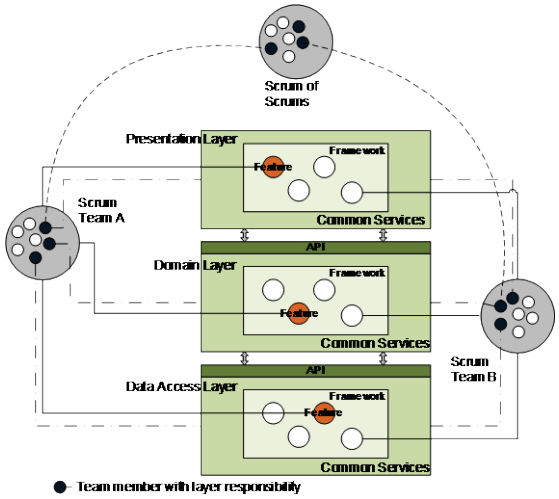

Fig. 6. Different teams assigned to features (vertical alignment), with some team members assigned to keep layers and frameworks consistent

In Fig. 3 we showed the teams organized primarily around the infrastructure. In Fig. 6, the teams have the necessary infrastructure to implement features quickly. Only a few team members, if any, have horizontal responsibilities. Yet every product development has to cope with changing requirements and new technologies.

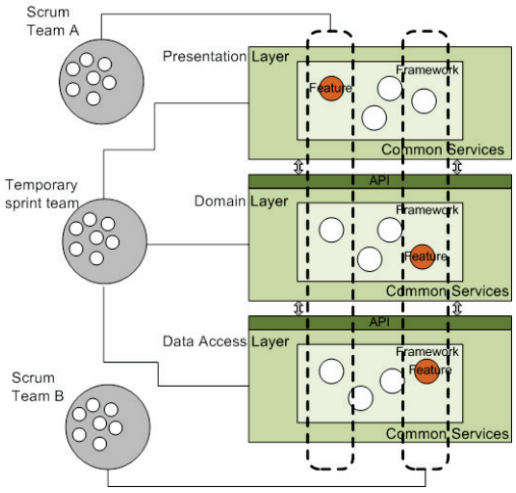

Fig. 7. Different teams assigned to features (vertical alignment), with a temporary team assigned to prepare layers and frameworks for future feature development

In Fig. 7 the teams now have primary responsibility for features. Some team members, including the product architect, look ahead to decide what will be needed in the

future. In one or more sprints, they dynamically self-organize into a temporary sprint team to develop the next piece of the runway, and then the team dissolves. Meanwhile, the other teams are organized vertically, developing features for the customer.

7 Conclusion

Architecture enables large-scale agile development. Key elements for success include focusing on architecture early and persistently throughout development, assigning an architecture owner as a counterpart to the product owner, and using the right architecting tools (e.g., tactics).

We contend that the issue is not to tweak individual agile practices to make them work outside of the agile sweet spot but to understand the specific issues of large-scale development, identify the problems that current practices cannot solve, and add architecture practices and tools. The goal for the software-development organization is to be agile at the level of the organization, not only in iteratively refining the architecture of the system under development but also in constantly tuning the development organization and improving the production infrastructure.

In this paper, we give an example of using architectural tactics and aligning architecture, agile development teams, and production infrastructure. A catalog of other tactics mapped to agile development can be collected from successful organizations and literature. Other ongoing and future work includes techniques to make architectural agility visible, identify and analyze architectural dependencies and incorporate dependency management into development, and provide timely feedback to support enhancement agility.

Acknowledgements. We thank the participants of the XP'2014 workshop in Rome on May 26 for their feedback and suggestions, in particular on Fig. 1.

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. This material has been approved for public release and unlimited distribution. DM-0001067.

References

1. Kruchten, P.: Contextualizing Agile Software Development. J. Softw. Evol. Proc. 25, 351–361 (2013)
2. Highsmith, J.A.: Agile Software Development Ecosystems. Addison-Wesley, Boston (2002)
3. Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R., Stafford, J.: Documenting Software Architectures. Addison-Wesley, Upper Saddle River, NJ (2011)
4. Perry, D.E., Wolf, A.L.: Foundations for the Study of Software Architecture. ACM SIGSOFT 17, 4, 40 (1992)
5. Meyer, B.: Agile! The Good, the Hype, and the Ugly. Springer, Zürich (2014)

6. McConnell, S.: Technical Debt, Software Best Practices. http://blogs.construx.com/blogs/stevemcc/archive/2007/11/01/technical-debt-2.aspx (2007)
7. Cockburn, A.: Walking Skeleton. http://alistair.cockburn.us/Walking+skeleton (1996)
8. Leffingwell, D.: Agile Software Requirements. Boston, Addison-Wesley (2011)
9. Brown, S.: Software Architecture for Developers. LeanPub, Vancouver, CA (2014)
10. Bellomo, S., Nord, R.L., Ozkaya, I.: A Study of Enabling Factors for Rapid Fielding: Combined Practices to Balance Speed and Stability. In: 35th International Conference on Software Engineering, pp. 982–991. IEEE Press, Piscataway, NJ (2013)
11. Nord, R., Ozkaya, I., Sangwan, R.: Making Architecture Visible to Improve Flow Management in Lean Software Development. IEEE Software 29, 5, 33–39 (2012)
12. Cataldo, M., Herbsleb, J.D., Carley, K.M.: Socio-Technical Congruence: A Framework for Assessing the Impact of Technical and Work Dependencies on Software Development. In: Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, pp. 2–11. ACM, New York (2008)
13. Nord, R.L., Ozkaya, I, Kruchten, P., Gonzalez-Rojas, M.: In Search of a Metric for Managing Architectural Technical Debt. In: Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture, pp. 91–100. IEEE Press, New York (2012)
14. Tamburri, D., Lago, P., Kruchten, P., van Vliet, H.: What Is Social Debt in Software Engineering? In: Sixth International Workshop on Cooperative and Human Aspects of Software Engineering, pp. 93–96. IEEE Press, San Francisco (2013)
15. Desbois, P. (ed.): Devops: A Software Revolution in the Making (Special Issue). Cutter IT J. 24, 8 (2011).
16. Shafer, A.C.: Infrastructure Debt: Revisiting the Foundation. Cutter IT J. 23, 36–41 (2010)
17. Bellomo, S., Kruchten, P., Nord, R.L., Ozkaya, I.: How to Agilely Architect an Agile Architecture? Cutter IT J. 27, 12–17 (2014)
18. Bachmann, F., Nord, R.L., Ozkaya, I.: Architectural Tactics to Support Rapid and Agile Stability. CrossTalk, 25, 3, 21–25 (2012)
19. Conway, M.E.: How Do Committees Invent? Datamation 14, 4, 28–31 (1968)
20. Cataldo, M., Herbsleb, J.D.: Factors Leading to Integration Failures in Global Feature-Oriented Development: An Empirical Analysis. In: 33rd International Conference on Software Engineering, pp. 161–170. ACM, New York (2011)
21. Cataldo, M., Herbsleb, J.D.: Coordination Breakdowns and Their Impact on Development Productivity and Software Failures. IEEE T. Software Eng. 39, 343–360 (2013)
22. Leffingwell, D., Martens, R., Zamora, M.: Principles of Agile Architecture http://scalingsoftwareagilityblog.com/wpcontent/uploads/2008/08/principles_agile_architecture.pdf (2008)
23. Bellomo, S., Kazman, R., Ernst, N., Nord, R.: Toward Design Decisions to Enable Deployability: Empirical Study of Three Projects Reaching for the Continuous-Delivery Holy Grail. In: First International Workshop on Dependability and Security of System Operation, pp. 32–37. IEEE Press, New York (2014)