Enabling Agility Through Architecture

Nanette Brown, Robert Nord, Ipek Ozkaya
Software Engineering Institute, Carnegie Mellon University

Abstract: Industry and government stakeholders continue to demand increasingly rapid innovation and the ability to adjust products and systems to emerging needs. Amongst all the enthusiasm for using Agile practices to meet these needs, the critical role of the underlying architecture is often overlooked.

Time frames for new feature releases continue to shorten, as exemplified by Z. Lemnios, Director of Defense Research and Engineering:

"Get me an 80% solution NOW rather than a 100% solution two years from now and help me innovate in the field" [1].

To meet these demands, government and government contractors are now looking closely into the adoption of agile practices [2] [3].

End users demand Enhancement Agility, the ability to keep adjusting the product to emerging needs through the addition of new features. Existing approaches to achieving Enhancement Agility vary, depending upon the lifecycle under which the product or system is being developed.

Under the Waterfall paradigm of software development, an extensive requirements phase is conducted to anticipate needs for both the initial and subsequent releases of the product or system being developed. Following the requirements phase, an architecture phase is conducted to develop a comprehensive underlying technical infrastructure. Within the Waterfall model, once the architecture is implemented, Enhancement Agility can be achieved, provided that the emergent user needs fit within the boundaries anticipated during the requirements phase.

However, taking the Waterfall approach presents two potential problems. First, when working in a new, unknown emergent problem space, building an architectural platform that reliably anticipates all future needs is an extremely difficult undertaking. Secondly, under the Waterfall paradigm, considerable effort and expense is incurred before any actual value is achieved (i.e., before any features are delivered to the user).

In contrast to Waterfall methodologies, Agile software development methods focus on delivering observable benefits to the end users through working software, early and often. A backlog of functional "user stories" is created. These stories are prioritized by end users and/or the product owner, acting as the user advocate. Development teams draw stories from the backlog and implement them in accordance with an end-user prioritization scheme. The Agile community's focus on continuous delivery of user-valued stories is another means of achieving Enhancement Agility. However, this approach also has its shortfalls, stemming mainly from an inadequate focus on dependency analysis.

Individual stories cannot be regarded in isolation. Stories have dependencies on other stories. In Software by Numbers, Denne and Cleland-Huang use the term "greedy algorithm" to refer to a prioritization scheme which focuses strictly on implementing the story with the highest immediate value [4]. They point out that, at times, higher-value stories may depend upon (i.e., require prior implementation of) lower value stories. Thus, truly optimizing value to the user requires teams to look ahead and anticipate future needs.

Similarly, stories have dependencies upon the architectural elements of the system. These dependencies exist regardless of domain stability or technical maturity. They exist regardless of whether the system is in its initial development stages or has been deployed and has been in the field for several years. The ability to identify and analyze architectural dependencies and incorporate dependency awareness into a responsive development model exemplifies the notion of Architectural Agility. It is our thesis that without Architectural Agility, Enhancement Agility cannot be reliably sustained.

Architectural Agility and Release Planning

Architectural Agility addresses shortcomings that occur within both the Waterfall and the Agile lifecycle models. Architectural Agility allows architectural development to follow a "just-in-time" model. Delivery of customer-facing features is not delayed pending the completion of exhaustive requirements and design activities and reviews. At the same time, Architectural Agility maintains a steady and consistent focus on continuing architectural evolution in support of emerging customer-facing features. It avoids the pitfalls of a myopic focus on user stories, which over time can lead to increased complexity and "tortured" implementation choices as develop-

ers seek to incorporate features that the architecture was not designed to support. Proceeding under the latter paradigm leads to the all-too-familiar situation in which features gradually take longer and longer to implement, the code becomes more and more buggy, and eventually management is informed that the system must be scrapped and rewritten "from scratch."

Our mantra for Architectural Agility is "informed anticipation." The architecture should not over-anticipate emergent needs, delaying delivery of user value and risking development of overly complex and unneeded architectural constructs. At the same time, it should not under-anticipate future needs, risking feature development in the absence of architectural guidance and support. Architectural Agility requires "just enough" anticipation. To achieve the quality of being "just enough," architectural anticipation must be "informed." Dependency analysis, real options analysis and technical debt management are the tools through which "informed anticipation" can be achieved. The remainder of this article will illustrate the application of these techniques through the practice of release planning.

Figure 1 shows a release planning board that represents the typical heuristics used within the Agile community for release planning. Desired stakeholder capabilities are represented as "user stories." These user stories are allocated to iterations in order of their priority to the end user.

Figure 2 shows an enhanced release planning board that incorporates planning for development of the underlying software architecture. In addition to selecting stories to be developed within each iteration, the team identifies the architectural elements that must be implemented to support them. This version of the release planning board also incorporates a "technical research" activity, recognizing that architectural development frequently requires investigation and analysis of alternate approaches. Finally, the term "capabilities" has been used in place of "user stories," reflecting a need to consider non-functional, quality attribute requirements, as well as the need to incorporate requirements across a broad range of stakeholders.

As an example, consider the Apps for the Army initiative [5]. The ability to add new and innovative apps quickly and easily exemplifies the concept of Enhancement Agility. However, Architectural Agility is required to supply the underlying technical infrastructure to support the app-based development model. The app-based development model includes a developer framework and run-time infrastructure that are part of the notion of an app store.

A conceptual architecture for an app store is illustrated in Figure 3. This conceptual architecture describes the essential high-level architectural elements such as content management, service management, data access, security and a range of external target devices that can access/manipulate the apps. Using an agile approach of starting small and growing the system, the team selects capabilities that support a small number of predetermined apps in the early iterations. This requires identifying those architectural elements within the business logic, data access, and service management com-

Figure 1: Agile iteration planning – focus on User Stories

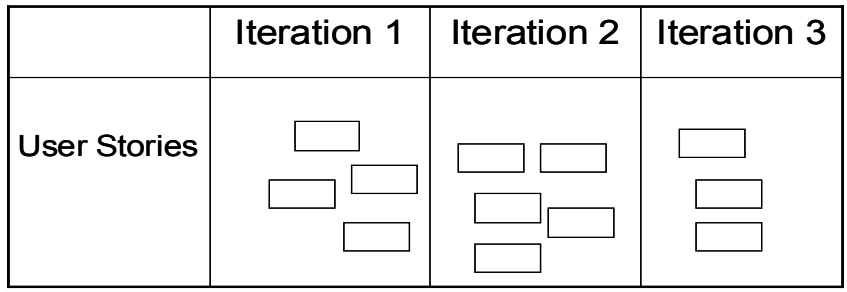


Figure 2: Architectural elements in agile iteration planning

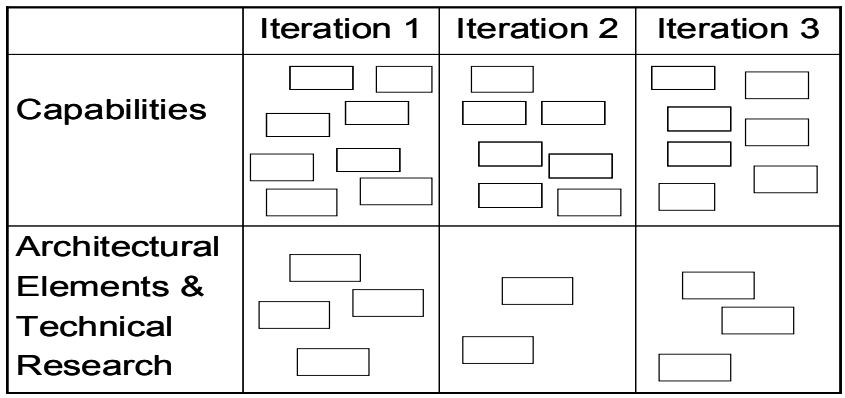
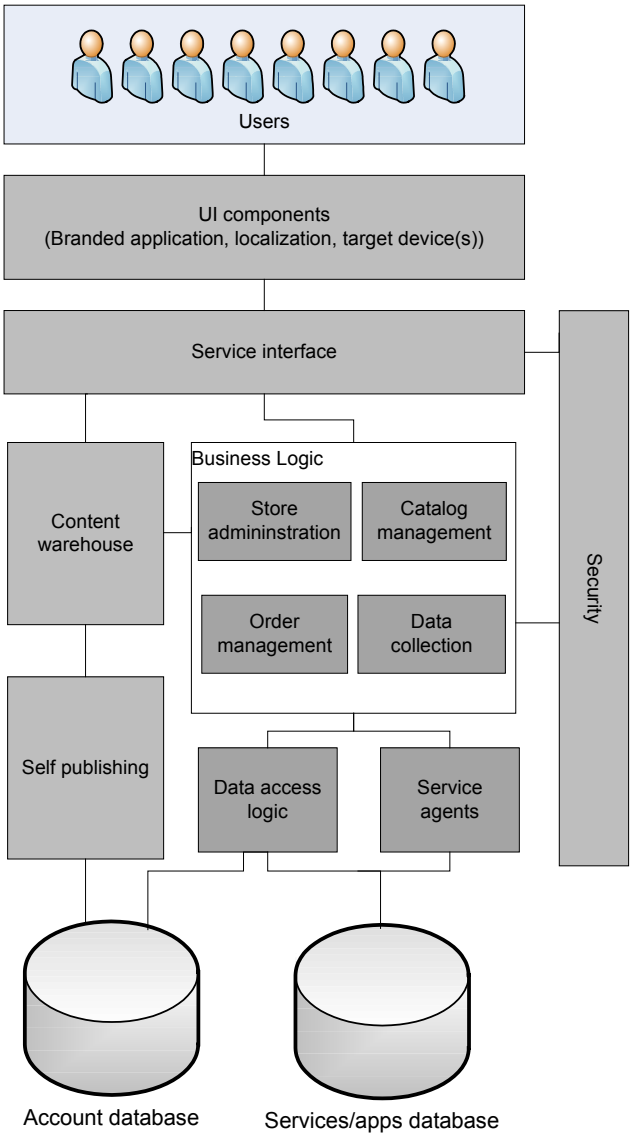
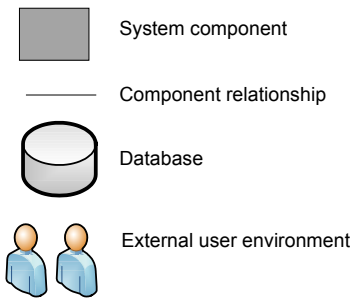


Figure 3: Conceptual App Store Architecture and High-Level Capability Dependencies



		Reporting	App catalog management	Sales management	Upgrade management	Promotion management	Order management	Account management	
Capabilities	Reporting		x	x	x	x	x	x	6
	App catalog management	x		x	x	x	x	x	6
	Sales management		x			x	x	x	4
	Upgrade management		x						1
	Promotion management	x	x	x			x	x	5
	Order management	x							1
	Account management						x		1
Architectural elements	UI components	x	x	x	x	x	x	x	7
	Service interface	x	x	x	x	x	x	x	7
	Content warehouse	x	x		x				3
	Self publishing	x	x		x				3
	Account database	x	x				x	x	4
	Services/apps database		x		x				2
	Data access logic		x				x	x	3
	Service agents		x		x	x	x	x	5
	Security		x	x	x	x	x	x	6
	Order management	x					x	x	3
	Data collection	x						x	2
	Store administration	x	x	x	x	x	x	x	7
	Catalog management	x	x		x				3
		9	11	4	9	5	8	9	



ponents that support these capabilities. In later iterations, the team expects to focus on scaling the system in the number of apps and users, enhancing security, and allowing users to contribute their own apps. Architectural elements within the security, content management, and publishing components need to be scrutinized to see which are needed to support these additional capabilities.

Implementing this type of planning heuristic requires the ability to do dynamic dependency management in a manner that is both rigorous and responsive. Dependencies between capabilities and architectural elements need to be identified for each iteration in order to prioritize and schedule work within a release.

Architecture Dependency Management

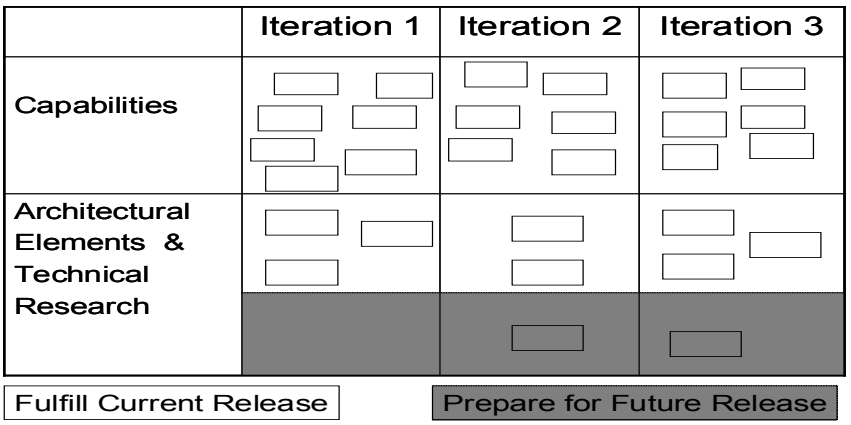
Dependency management has been studied extensively at the level of code artifacts. Applying dependency management at the architecture level is beginning to show promising results due to increasingly effective tool support. These metrics can be extracted from the architecture, represented in the form of a Dependency Structure Matrix (DSM). The DSM is a compact representation which lists all constituent subsystems/activities and the corresponding information exchange and dependency patterns. Domain Mapping Matrices (DMMs) augment DSM analyses and can be used to represent the dependencies between capabilities and architectural elements.

Returning to the example, dependency analysis for the app store must consider dependencies between capabilities

as well as dependencies between architectural elements and capabilities. These dependencies are identified in the matrix in Figure 3. The capabilities portion of this matrix is an example of a DSM. An X mark indicates that the capability in the row provides information to the capability in the column. Reading across the row labeled "App catalog management," it is clear that all other capabilities depend on it. The architectural elements portion of the matrix is an example of a DMM. A marked cell indicates that the architectural element in the row implements an aspect of the capability represented in the column. Reading down the column labeled "App catalog management," it becomes clear that the App catalog management capability depends on almost all of the architectural elements. Having this kind of view can be essential in focusing the iterations within releases.

Metrics associated with dependency also provide data for inferring the likely costs of change propagation, especially when dependencies between architectural elements are also considered (not shown in Figure 3). One such example is discussed in Carriere et al where the value of re-architecting decisions needed to be understood to determine if the expense to implement them was justified [6].

Figure 4: Informed anticipation in the context of agile release planning



Architecture Heuristics Focused on Value: Real Options Analysis and Technical Debt Management

For effective Architectural Agility, dependencies between capabilities and architectural elements need to be identified not only to fulfill the current release, but to plan for future releases as well (Figure 4). Informed anticipation requires incorporating architecture heuristics focused on value into the planning model. Real options analysis and technical debt management offer potential models to make an informed choice and find the right balance of agility, innovation, and speed on the one hand, and governance, flexibility, and planning for future needs on the other.

This additional set of considerations adds a new dimension to the release planning board. This added dimension allows the identification of architectural constructs that, while not required for the current release, should potentially be incorporated into the current release in anticipation of future stakeholder goals.

As an example, the initial number of deployed apps is expected to be small, so capabilities such as scalability could be deferred and assigned to a future release. However, it is also true that by setting up an app store scalability infrastructure—that is, buying the option of scaling up—you can reduce your technical debt down the road. By choosing to take a shortcut—not buying the option—you incur possible technical debt.

The question of how to optimally allocate architectural elements that deal with scalability to releases can benefit from applying real options analysis. Real options analysis is a financial analysis model to help determine whether some upfront cost should be spent (buying the option) to have the right, but not the obligation, to take an action in the future (exercising the option). The real options analysis method applies the financial options theory to quantify the value of flexibility in real assets and business decisions to determine the value of such delayed decision making. And both common sense and the theory demonstrate that the higher the uncertainty, the more it makes sense to wait to act and defer the decisions. From this perspective, the agile community has used the concept of real options in separating concerns that have immediacy and those that can possibly wait.

In agile release planning, real options analysis is a way to look at the allocation of architectural elements to releases based on their dependencies from the perspective of future value [7]. In architecture terms, taking an option could be applying an architecture pattern, providing a well-structured modular design that supports Enhancement Agility. Real options analysis can be informed and complemented by a consideration of technical debt.

The technical debt metaphor [8] highlights that doing things the quick and dirty way for short-term benefit sets us up with a technical debt. Like a financial debt, the technical debt incurs interest payments, which come in the form of the extra effort that we have to do in future development because of suboptimal design choices. We can choose to continue paying the interest, or we can pay down the principal by refactoring and improving the design. Although it costs to pay down the principal, we gain by reduced interest payments in the future.

Agile development methods aim to manage technical debt through refactoring practices. Refactoring is restructuring an existing body of code, altering its internal structure without changing its external behavior. However, when significant architectural change is needed, such small, local refactoring efforts cannot compensate for the lack of an architecture that is necessary to guide the architect in achieving the goals of the system. In this case, lack of Architecture Agility starts compromising Enhancement Agility.

Informed Anticipation Guiding Agile Release Planning

Unifying the concepts of technical debt, real options, and uncertainty management is a common focus on the question "Should I take a certain action today in anticipation of increased benefit and reduced cost in the future?" Taking the correct action today provides an option which can be acted upon in the future. This is where the agile mindset and architecture reasoning tend to diverge. Agile projects focus on stories that are needed in the current release and rely on code-level refactoring to incorporate future stories. However, relying only on code-level refactoring often does not suffice, especially in large-scale development.

Spending some time architecting can provide better options in many large-scale development contexts that struggle with applying agile techniques. The cost and benefit tradeoff is often misrepresented as a choice between "do nothing" and "spend a lot of time on something you may not need." The concrete benefit of having real options requires the tradeoff to be made between "do nothing, possibly suffer a lot later" and "do just a little, suffer less later."

Identifying architectural elements that enable future stakeholder goals requires mapping options to releases across the lifespan of the system. A real option often requires some portion of the system to be developed today to enable future development at ease. Understanding which release that option needs to be allocated to and how its cost will be paid during that release are key to success. The release planning board provides a visual means to monitor such elements throughout the releases. Although lower in cost, options are not without expense, so there should not be too many. But cost is not the only issue, so a large-scale project without any options should be viewed with a critical eye. Ideally, the decision to develop an option should be justified by the desire to mitigate the risk of an uncertain future.

Identifying dependencies within a given release also requires understanding the deliberate shortcuts taken to achieve the high-priority functionality. These shortcuts (technical debt) need to be revisited at each iteration. Monitoring these decisions is the first step to realizing the good enough, but cost effective solution today without endangering the needed full solution tomorrow. Once identified, the decision can be made at appropriate times to emphasize more architecting and paying off the debt as opposed to adding new features.

Looking back at the conceptual architecture shown in Figure 3, even at this level, several decisions can be made by taking advantage of dependency analysis in relationship to real options and technical debt concepts. The App catalog management capability describes the feature allowing users to author and add apps to the app store. The matrix shows that the Self-publishing component has a role in implementing this feature. Depending on the cost and value of early delivery versus the level of control, two approaches are available. In a quick delivery approach, rather than implement the full functionality in a separate Self-publishing component, initially a subset could be implemented in the Store administration component that has been selected for implementation in an early release for other reasons. Administrator users have full access to this component through the Sales management capability. This approach would depend on the administrator to ensure that only authorized and well-behaved apps are published, but since this approach limits exposure of the infrastructure and is simpler to implement, it could be deployed quicker. In conjunction with this approach, preparing for the future release and creating the infrastructure for self publishing can be an option for future investment. When the time comes, the infrastructure could be self enabled, increasing the innovation of apps by allowing users to submit their own without external controls.

Technical debt is most often associated at the level of detailed design and code artifacts and tool support is beginning to show promise [9]. An analog for monitoring and managing technical debt in the architecture would provide analyses earlier in the development cycle for keeping the project on track. Some of these measures exist and can be used today. For example, Hinsman from L.L. Bean [10] used a tool to analyze and monitor architecture violations based on dependency analysis in an ongoing effort to evolve and improve its architecture. Once the architecture was restructured, the process was modified to support agility through keeping the architectural elements visible so that they could be explicitly managed.

Key Take-Aways

A focus on architecture is not in opposition to Agile values and principles. In fact, ongoing sustainable achievement of Enhancement Agility is only possible when coupled with Architectural Agility. To achieve Architectural Agility, the Agile community must first expand its focus on end user stories and address the broader topic of capabilities, including quality attribute requirements and a diverse range of stakeholders. The use of dependency analysis practices can be used to facilitate a "just-in-time" approach to building out the architectural infrastructure. Real options and technical debt heuristics can be used to optimize architectural investment decisions by analyzing uncertainty and tradeoffs between incurred cost and anticipated value.

ABOUT THE AUTHORS

The authors work in the Research, Technology, and System Solutions Program at the Software Engineering Institute and are currently engaged in a research project on "Communicating the Value of Architecting within Agile Development."

Nanette Brown is a Visiting Scientist and is a Principal Consultant with NoteWell Consulting. She is engaged in activities focusing on architecture within an Agile context. Previously, Nanette worked at Pitney Bowes Inc., most recently as Director of Architecture and Quality Management, where she was responsible for design and implementation of a customized SDLC that blended RUP and Agile practices.

Robert L. Nord is a senior member of the technical staff and works to develop and communicate effective methods and practices for software architecture. He is co-author of the practitioner oriented books, Applied Software Architecture and Documenting Software Architectures: Views and Beyond, published by Addison-Wesley and lectures on architecture-centric approaches.

Ipek Ozkaya is a senior member of the technical staff. Her primary work is on developing techniques and methods for improving software architecture practices by focusing on software economics and requirements management. Currently, she serves as the technical lead of the agile development and software architecture independent research work, in addition to leading work in architecture-based system evolution. In addition she contributes to teaching of several courses in the Software Architecture Certificate Program at the SEI.

Contact Information

Nanette Brown
nb@sei.cmu.edu

Robert L. Nord
rn@sei.cmu.edu

Ipek Ozkaya
ozkaya@sei.cmu.edu

Software Engineering Institute

Carnegie Mellon University

4500 Fifth Avenue
Pittsburgh, PA 15213-3890
Tel: +1 (412) 268-7700
Fax: +1 (412) 268-5758
URL: http://www.sei.cmu.edu/architecture/

REFERENCES

1. Lemnios, Z. (2010) Statement of Testimony Before the United States House of Representatives Committee on Armed Services Subcommittee on Terrorism, Unconventional Threats and Capabilities, March 23, 2010. [cited on June 11, 2010] URL: <http://www.dod.mil/ddre/Mar232010Lemnios.pdf>
2. Cohan, Sean (2007) *Successful Integration of Agile Development Techniques within DISA*, AGILE 2007.
3. Crowe, P, Cloutier, R. (2009) "Evolutionary Capabilities Developed and Fielded in Nine Months," *CrossTalk*, May 2009. URL: <http://www.stsc.hill.af.mil/crosstalk/2009/05/0905CroweCloutier.html>
4. Denne, M., & Cleland-Huang, J. *Software by Numbers: Low-Risk, High-Return Development*. Upper Saddle River, N.J.: Prentice Hall. 2004.
5. CIO/G-6 Public Affairs. G-6 launches 'Apps for the Army' challenge. [cited on June 11, 2010] URL: <http://www.army.mil/-news/2010/03/01/35148-g-6-launches-apps-for-the-army-challenge/>
6. Carriere, J. Kazman, R., Ozkaya, I. "A Cost-Benefit Framework for Making Architectural Decisions in a Business Context" in Proceedings of the 32nd International Conference on Software Engineering, Vol 2, pp:149-157, 2010
7. Bahsoon, R., Emmerich, W., Macke, J. "Using Real Options to Select Stable Middleware-Induced Software Architectures." *IEE Proceedings Software* - Special issue on relating software requirements to architectures 152(4) (2005) ISSN 1462-5970, pp. 153-167, IEE press.
8. Fowler, M. *Technical Debt Quadrant*. Bliki [Blog] 2009 [cited on June 14, 2010]; URL: <http://www.martinfowler.com/bliki/TechnicalDebtQuadrant.html>
9. Gaudin, O. Evaluate your technical debt with sonar, [cited on June 11, 2010] URL: <http://www.sonarsource.org/evaluate-your-technical-debt-with-sonar>
10. Hinsman C., Sangal, N., Stafford, J. *Achieving Agility Through Architecture Visibility*, in LNCS 5581/2009, Architectures for Adaptive Software Systems, 2009 pp.116-129