Software Engineering Institute
Carnegie Mellon University

Enabling Incremental Iterative Development at Scale: Quality Attribute Refinement and Allocation in Practice

Neil Ernst
Stephany Bellomo
Robert L. Nord
Ipek Ozkaya

June 2015

TECHNICAL REPORT
CMU/SEI-2015-TR-008

Software Solutions Division

http://www.sei.cmu.edu

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 report was prepared for the
SEI Administrative Agent
AFLCMC/PZM
20 Schilling Circle, Bldg 1305, 3rd floor
Hanscom AFB, MA 01731-2125

DM-0002248

Table of Contents

List of Figures

List of Tables

Abstract

Lengthy requirements, design, integration, test, and assurance cycles delay delivery, resulting in late discovery of mismatched assumptions and system-level rework. In response, development methods that enable frequent iterations with small increments of functionality, such as agile practices, have become popular. But such methods de-emphasize architectural analysis; they assume the emergence or existence of a stable architecture. Yet as the business goals and context evolve, the architecture must also change, which requires allocating increments of quality attribute requirements to iterations along with other business capabilities. Quality attribute requirements (also called nonfunctional requirements) are hard to separate into smaller increments since they often crosscut many aspects of the product. As a result, allocation is uneven since it is challenging to decompose them and understand their value. Working with quality attribute requirements in an incremental and iterative fashion involves solving two problems: separating high-level requirements into their constituent parts and allocating them to iterations to fulfill the requirement. Underpinning both problems is the need for measurements to show that the requirement is satisfied. This report describes industry principles and practices used to smooth the development of business capabilities and suggests some approaches to enabling large-scale iterative development, or "agile at scale."

1 Introduction to Quality Attribute Requirements

Lengthy requirements, design, integration, test, and assurance cycles delay delivery, resulting in late discovery of mismatched assumptions that, in turn, result in system-level rework. In response, software development teams have turned to methods that enable frequent iterations with small increments of functionality, such as agile practices. But these methods are light on architectural analysis; they assume the emergence or existence of a stable architecture. Yet as the business goals and context evolve, the architecture must change as well, which requires allocating increments of quality attribute requirements to iterations along with other business capabilities.

Quality attribute requirements (QARs) are qualifications of the functional requirements of the overall product [Bass 2012]. The overarching aim in dealing with quality attributes is to ensure that the system satisfies the criteria of interest to judge the quality of a system's operation, rather than specific behaviors. For example, building a system to have "good performance" means understanding what "performance" is and what "good" means. In iterative software development processes (such as Scrum), a requirement such as "improve performance" carries implications for the design and infrastructure of the system that will often span multiple iterations. Consequently, a development team needs to refine high-level QARs into subparts that are iteration sized. In doing so, they must answer two questions:

1. What are the important tangible **refinements** of the QAR, and how do they relate to each other? This is both an analysis and a design activity.
2. How are the parts **allocated** to iterations in the software development process? This is a process management activity.

Underpinning each question is the concept of measurement—to show that the requirement and its cost–benefit tradeoffs are satisfied.

QARs, also known as nonfunctional requirements, are particularly hard to refine into smaller increments since, by their nature, they tend to crosscut many aspects of the product. As a result, allocation is uneven since it is challenging to break QARs apart and understand their value. In this report, we identify common practices for refining and allocating software development work that focus on QARs in iterative (or agile) software development processes. In these settings, there is a tendency to focus on functional deliverables at the expense of QARs: "Customers often focus on core functionality and ignore NFRs [QARs] such as scalability, maintainability, portability, safety, or performance" [Cao 2008]. As a result, costly rework is often required to correctly satisfy the QAR [Brown 2011].

We discuss some mechanisms to eliminate rework related to QARs. First, we introduce quality attribute–focused software development work; then we discuss practices described in the existing literature that deal with implementing QARs, from refining QARs, to allocating them to iterations, to using existing industry practices to manage iterations. This overview offers development teams a number of ways to overcome challenges to large-scale incremental iterative development, or "agile at scale."

2 Coping with Quality Attributes in Iterative Development

Figure 1 shows the basic concepts of a software development process in which the role of QARs in a software development project can be understood. QARs are refined from business goals (on the left), an architecture is proposed to satisfy them, the implementation tasks (on the right) are allocated, and then the system is implemented based on conforming to the architecture. Over time the system evolves to satisfy the business goals.

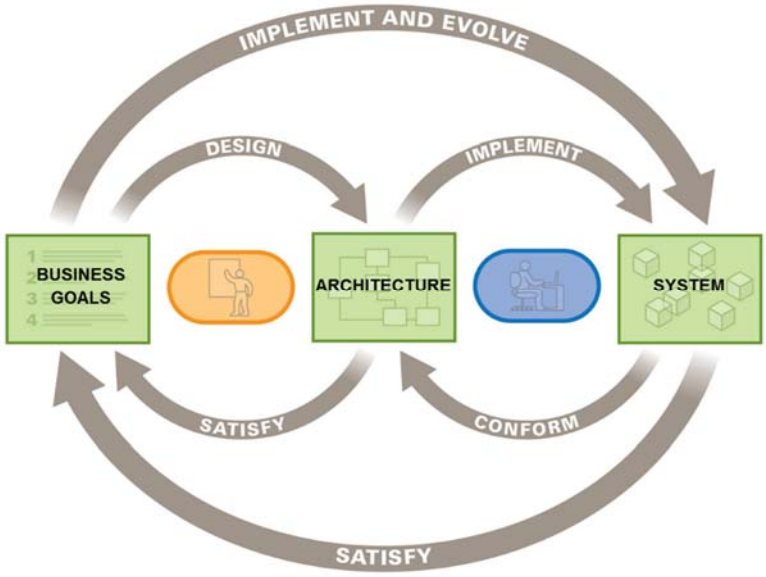

Figure 1: Architecture-Centric Engineering for Improving Software Systems

In this report, we look at how QARs are dealt with in highly iterative development. Almost always this means refinement and allocation are never fully completed since the architecture and implementation are continuously evolving. We must return to the business goals at the end of each iteration to reexamine how well they are satisfied by what we learned in designing and implementing the solution. This process suggests a need to view software design as a series of evolving decisions at varying levels of abstraction: selecting a programming language, deciding on a reference architecture (e.g., *n* tier), using frameworks [Cervantes 2013], understanding modules, and selecting tactics and patterns to guide implementation.

The advantages of using an evolutionary approach have been well documented [Breivold 2012]. When we talk about refining QARs or allocating them to iterations, we do not suggest that a development team performs either task only once. Projects can have many different durations for iterations and increment planning, depending on the software development context, ranging from weeks to years. At one end of the spectrum, allocation may involve pulling from a backlog user stories that fit the iteration length (e.g., one- or two-week sprints in Scrum). At the other end of the spectrum, allocation of QARs might be pulled from a Statement of Work or contractual agree-

ment detailing the work breakdown structure for the next two or more years. Neither scenario implies the completion of architecting activity that drives the allocation of QARs before development starts, although the latter has wrongly been assumed to imply so.

A QAR focus implies an effort to improve the state of satisfaction of the QAR. For example, an iteration may be heavily weighted toward improving performance or a reengineering activity to change web frameworks in response to a need for more flexibility. QARs and the work associated with them are not as independent as features, and the development team must carefully consider dependencies when packaging the pieces into units that must be allocated together or sequenced over iterations and releases [Nord 2014, Sethi 2009].

As noted by Bachmann, Bass, and Klein, "the key for design is characterizing the set of changes that a particular system will be subjected to" [Bachmann 2003]. Bachmann and colleagues argue that there are four challenges in moving from QARs to design:

1. precisely specified QARs
2. enumeration of design approaches that achieve QARs
3. linkage between the QAR and the design fragments that achieve it
4. a method for composing the various design fragments into a cohesive whole

We note two additional challenges. One challenge is to correctly specify the QAR to a level of detail that is achievable in some limited period of time (the release cadence) and to find design fragments that can be completed in one iteration (where an iteration is a segment of the release cadence). We call this the *refinement problem*. The other challenge is an *allocation problem*: to correctly allocate design fragments to iterations to optimize the relationship between cost and value. This relationship is a complex one. In some situations, deferring implementation may lead to a cost of delay, such as failing to introduce a caching system to support a holiday promotion component and avoid outages due to increased traffic [Anderson 2012, Slide 37]. In other situations, implementation choices made to meet the constraints may lead to costly rework.

3 Refining Quality Attribute Requirements

QARs are often provided in unstructured and unclear ways. QAR refinement is the process of elaborating and decomposing a QAR into a specifiable, unambiguous form achievable in one release[1] as well as finding design fragments that will accomplish that task. In the agile literature, this is also known as slicing or sizing and is typically applied to user stories. In the requirements literature, this has been called the Twin Peaks model [Nuseibeh 2001] because one iteratively progresses down the peaks of requirements and architecture.

Refinement of QARs ideally results in a unit of work small enough to be testable, small enough to fit in an iteration, and useful enough to produce value. Getting to the appropriate size is a process of analysis and synthesis, separating abstract requirements into constituent parts so they and their interrelationships can be studied, combining constituent parts into a unified entity that meets the criteria. It is a design activity: "it surfaces our ignorance of the problem domain" [Khan 2014]. Design support for any one quality attribute will need to be traded off against design support for realizing functional requirements and other QARs, fixing defects, making infrastructure investments, and so on. Identifying measures for the quality attributes of interest is essential input in guiding the design activity, making compromises among competing concerns, and scoping incremental improvements.

There are a number of ways to size requirements, some based purely on analyzing the requirement, some based on the work involved in satisfying the requirement, and some based on a mixture of analyzing the problem and possible design fragment that contribute to the solution. We have highlighted approaches to sizing requirements in Table 1.

Table 1: Approaches to Refinement

Approach	Description	Source
Vagueness	Break down vague terminology such as *manage*	Green 2013 Lawrence 2009
And/Or decomposition	Split on conjunctions *And Or*	Antón 1996 Green 2013
Acceptance or test criteria	Satisfy one criterion per slice	Green 2013
Workflow/use case steps	Use one slice per step; frequently seen as an anti-pattern	Adzic 2012 Green 2013 Lawrence 2009 Verwijs 2013
Business rule	Use one slice per variation in a rule	Cervantes 2013 Verwijs 2013
Dependencies	Use one slice per dependency	Denne 2003 Lawrence 2009
User interface (UI) alternatives	Classify by input (e.g., keyboard vs. mouse selection) or output (e.g., screen size)	Lawrence 2009 Verwijs 2013

[1] A *release* is delivery of the software to the broad customer base; an *iteration* is a sequence of time during delivery, which often produces working software that is not necessarily widely released.

Approach	Description	Source
Ratcheting	Ratchet quality attribute criteria, such as “works” vs. “works fast”	Gilb 2007 Humble 2010 Kua 2013 Lawrence 2009 Wirfs-Brock 2011
Prototype/spike	Add spikes where steps are unknown; “investigate” vs. “implement”	Lawrence 2009
Horizontal slicing	Slice according to architectural layers (various); commonly seen as an anti-pattern (cake metaphor) since it silos the work	Verwijs 2013
Path based	Split by normal/happy and problem paths through the application	Verwijs 2013
Parameters	Use input parameter options such as phone number, zip code, etc.	Verwijs 2013
Operation type	Classify by Create, Read, Update, Delete, etc.	Verwijs 2013
Roles	Classify by user vs. admin	Verwijs 2013
Use case	Set the overall parameters via use cases; slice the use cases with user stories	Cockburn 2008
Hamburger slicing	Create horizontal slices that map steps in use cases; then extend vertically according to improved quality criteria	Adzic 2012

Not all of these approaches require lengthy elaboration, but a few deserve more detailed explanation.

3.1 Ratcheting

The techniques of *ratcheting* and *acceptance* or *test criteria* use quantifiable outputs of each iteration to identify new opportunities for work. For example, Iteration 1 asks for the system to work, Iteration 2 for the system to be faster, and Iteration 3 for it to be as fast as possible. There is a danger that over-operationalizing might lead to poor comparisons: improving a system to be twice as fast might involve much more than twice the work. Methods such as set-based design [Kennedy 2014], Planguage [Gilb 2007], and landing zones [Wirfs-Brock 2011] all leverage this idea of progressively ratcheting user story targets to improve quality attribute response. A more elaborate form of ratcheting may involve the other components of a user story or quality attribute scenario, for example, changing the source of a request from internal to external actors.

3.2 Horizontal Slicing

The *horizontal slicing* and *workflow/use case* techniques amount to identifying the steps of the use case, scenario, or user story and assigning each phase to a requirement. For example, the login screen, user validation, database query, and business logic might all be done one after the other. However, many think this is counterproductive. For one, it prevents end-to-end testing from working. Walking skeleton or “tracer bullet” approaches [Basili 1975, Cockburn 1996, Hunt 1999], in which one builds a very simple application demonstrating complete (if simplistic) functionality, at least allow for proper tests. Furthermore, focusing on one horizontal slice (such as the UI) can lead to premature optimization (for example, if the database layer changes afterward).

Approaches including *hamburger slicing* [Adzic 2012], splitting by *operation type*, and *business rule*, on the other hand, allow one to show how the software system can support the entire scenario.

3.3 Prototypes and Spikes

One motivation for adopting an incremental approach is to favor "responding to change over following a plan" [Beck 2001]. *Prototypes* and *spikes* are information-gathering activities that uncover unknowns about a design problem [Leffingwell 2011b]. After all, we can implement only what we know. Use of prototypes or spikes might be triggered if the project has mounting technical debt or wide variation in cost and effort estimates. In some cases, a spike becomes a specific requirement, and "learn more about X" is assigned to a developer or architecture team. The spike might take the form of an experiment or design expansion, for instance, by using A/B testing. One approach is to do prototyping with a specific QAR focus, as described by Bellomo and colleagues [Bellomo 2013]. Typically prototype work does not lead to code that is released.

3.4 Goal Elaboration

One can use techniques such as the Goal-Based Requirements Analysis Method (GBRAM) to perform refinement on higher level QARs using *And/Or* slicing [Antón 1996]. Developers begin top-down refinement by asking how the top-level goal might be achieved in some combination of subgoals. This approach lends itself to formal analysis, and modeling and development teams often prefer it in settings where detailed safety or hazard analysis is desired. For example, search algorithms can recommend optimal (e.g., minimal) solutions to the top-level goals. In a variation on this approach, Gottesdiener and Gorman break high-level or abstract constraints along six dimensions for expansion: user, actions, data, business rules, interfaces, and quality attributes [Gottesdiener 2010]. This creates a much larger yet better described epic. In particular, they advise exploring options for common quality attributes (security, performance, etc.) and then using Planguage tags to anchor the expanded user story to a measurable outcome [Gilb 2007].

3.5 Empirical Evaluation

In a related paper, we described two case studies that we conducted with software companies [Bellomo 2014b; Ernst 2014]. We investigated how these firms managed architectural work and found that ratcheting was the primary approach. We observed that developers refined performance requirements using a feedback-driven approach, which allowed them to parse the evolving performance requirements, expressed as state transitions, to meet increasing user expectations over time. Within each state transition, developers refined crosscutting concerns into requirements by breaking them into their constituent parts in terms of the scope of the system and response to stimuli in a given context. The system and crosscutting performance requirements evolve as the stimuli, context, and response are ratcheted.

We see evidence of projects that are better able to sustain their development cadence with a combination of refinement and allocation techniques guided by measures for requirement satisfaction, value, and development effort. As we retrospectively analyzed these examples, we found that these teams did not follow a formal technique; however, they did have common characteristics in how they refined the work into smaller chunks, enabling incremental requirements analysis and allocation of work into implementation increments.

4 Allocating Quality Attributes to Iterations

The purpose of sizing work is to allocate pieces of work to iterations. In an agile process, allocation involves taking stories from the product backlog and adding architectural stories directly to the iteration backlog [Gottesdiener 2010]. This does not mean that the task breakdown is planned months in advance. The backlog contains stories that one would classify as specific to a particular quality attribute and design fragments associated with the quality attribute. Publically available examples of stories can be seen in industry-relevant open source systems. For example, the CONNECT and HADOOP Distributed File System (HDFS) projects have many user stories that deal with performance metrics and documentation [CONNECT 2014, Apache 2014]. The advantage is that architectural slices have high visibility; the disadvantage is that these stories may be poorly thought out and slip off the backlog.

Figure 2 illustrates several allocation process patterns. These patterns summarize how different software development life cycles can be generalized with respect to how they balance architecting with feature development. We describe these patterns and provide key questions that the development team should address in allocating QARs within the boundaries of that pattern.

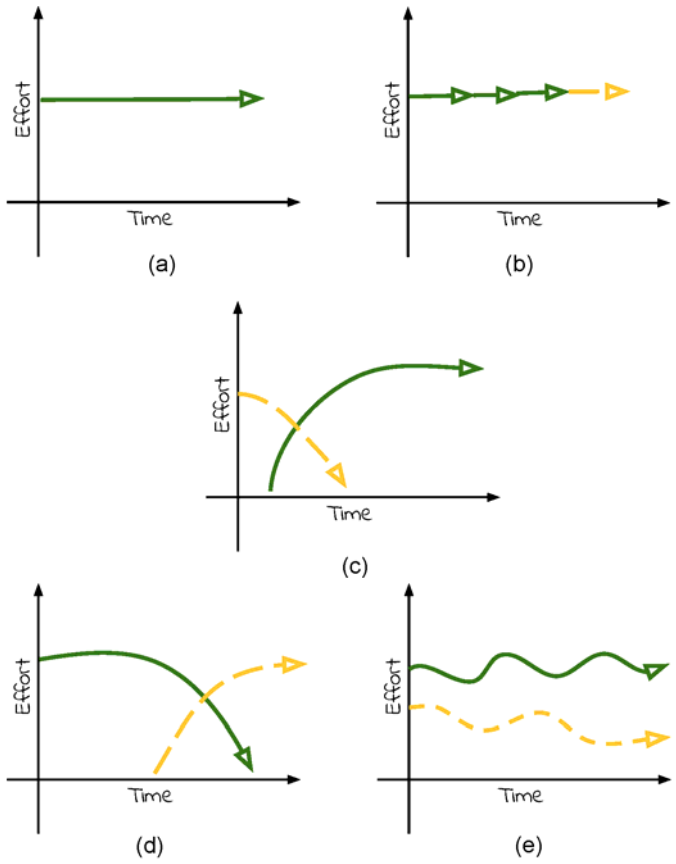

Figure 2: Allocation Process Patterns

Solid green lines indicate feature/functional work, dashed yellow lines architectural work. (a) No architectural work (YAGNI); (b) hardening sprints; (c) Iteration Zero; (d) rework; (e) evolutionary/runway.

4.1 YAGNI

In Figure 2(a), the "You Aren't Going to Need It" (YAGNI) practice ignores any architectural work as non-value added, which is why the pattern image has no architectural line. To be fair, YAGNI does not mean never do architectural work but rather evolve and refactor it [Séguin 2012]. Pure versions of this method are more evolutionary (see Section 4.5). However, in practice it has been interpreted to mean "never do design." YAGNI appears to be a popular approach because of the significant upfront savings in architecture effort. However, taking shortcuts can result in significant re-architecting effort and rework later on [Bellomo 2014a].

Key Questions	*Will you* ever *"need it"? How does the development effort value architectural work during the current increment?*

4.2 Hardening Sprints

Traditionally, a hardening sprint, shown in Figure 2(b), has been defined as the final sprint prior to release that is dedicated to fixing bugs and improving QAR satisfaction. This is called an *anti-pattern* since these bugs should have been fixed as part of the work that introduced them. However, development teams considering how to scale agile use this term to group together production-release tasks that only occur immediately prior to release, such as in Leffingwell's Scaled Agile Framework (SAFe) [Leffingwell 2014b] or the Disciplined Agile Delivery method's Transition Phase [Ambler 2012]. Leffingwell also uses this phase for verification and validation activities in high-assurance environments [Leffingwell 2011a]. You can see an example in CONNECT Sprint 120 [CONNECT 2013]. A variation of this pattern is called *cleanup*, in which rather than cleaning up the code prior to release, the code is first released and subsequent architectural work is done to clean up the code base. Galen gives an overview of terminology, including release readiness, stabilization, or spring cleaning sprints [Galen 2014]. Dedicating a single sprint to cleanup means that the entire team is engaged, though such a sprint makes a good target for cutting if time is a factor. Architectural issues are not always amenable to deferral.

Key Questions	*How often are hardening sprints needed? How many are enough?*

4.3 Iteration Zero

In Figure 2(c), Iteration Zero involves architecture planning before writing any code. An overly long Iteration Zero is equivalent to the dysfunctional "Big Up-Front Design" (BUFD) anti-pattern. Meta-methods like the Rational Unified Process [Leffingwell 2011b] and Disciplined Agile Delivery [Ambler 2012] include a preliminary design phase, which may itself be iterative and which Kruchten calls "architectural iterations" [Kruchten 1998]. Logically every development effort needs at least some degree of initial envisioning or a well-understood platform with a well-defined architecture to start from. Architectural Iteration Zero is specifically about deciding on some important architectural properties, including frameworks and patterns. Figure 2(c) captures this concept in the length of time before the amount of work on features and functionality (solid green line) exceeds the work done on architecture and planning (dashed yellow line). Variations of Iteration Zero capture tradeoffs between over-analysis and rework, best illustrated by Boehm's "sweet spot" discussion [Boehm 2003].

Iteration Zero fits with well-established approaches to planning and project management, particularly in government contexts. It provides an opportunity to take a high-level view of the problem before becoming enmeshed in low-level work. However, it is possible to succumb to analysis paralysis. It is also unlikely that a development team can understand all ramifications of decisions prior to feedback from experimentation and implementation. While Iteration Zero implies an activity that happens before the rest of the design and development effort, the essence of the approach is the focused planning and design activity. A large-scale development effort may have several Iteration Zeros for different parts of the system.

Key Questions	*How long should Iteration Zero be (i.e., what amount of design is necessary vs. wasteful)? When should Iteration Zero be revisited for different parts of the system?*

4.4 Rework

As shown in Figure 2(d), rework is more of an anti-pattern, wherein feature development comes to a screeching halt as technical debt becomes impossible to ignore [Nord 2012], and substantial portions of the codebase must be rebuilt. Typically this approach is also associated with substantial rework costs. When architecture is accounted for up front, it can allow for rapid exploration of alternatives to understand and manage rework, but this approach costs time and effort, and it delays other feature work.

Key Questions	*When are evolutionary architectural improvements still possible within the release cadence? When does development cross the boundary that necessitates rewriting the system (disrupting development)?*

4.5 Evolutionary/Runway

A runway, shown in Figure 2(e), "exists when there is sufficient system infrastructure in place to allow incorporation of near-term product backlog" [Leffingwell 2008]. It emphasizes a low level of ongoing architectural work that comes and goes, hence the wavy pattern. It differs from agile approaches in which user stories for architecture are assigned to the product backlog as a whole; runways tend to be separate work products and are more common in larger projects.

A related approach is vertical slicing, where one builds out the runway based on which stories need a given architectural element [Brown 2011]. In the work of Denne and Cleland-Huang, vertical slicing is managed with a dependency graph: one first decides which minimally marketable features to prioritize and then identifies the common architectural elements to those high-priority features [Denne 2003]. Properly sequenced, the runway team can act in coordination with development to evolve the architecture needs. However, a runway team may potentially lose contact with the state of development in the main implementation branch.

Key Questions	*How does the development team identify opportune moments and opportune places for minor re-architecting improvements? How long should the runway be?*

4.6 Edge Cases

The final patterns are better described as edge cases. In Figure 3(f), both feature and architecture work increase over time, as one might see as a product scales up to meet increasing demand. In Figure 3(g), the opposite holds, for example, if a project has reached the end of its life cycle or work on a version has been completed. These are simple and easily understood patterns that often result from a project being overcome by events. However, features and architecture may not grow or shrink at the same rate.

Key Question	*At what point do the levels of effort start to change (i.e., what is the inflection point)?*

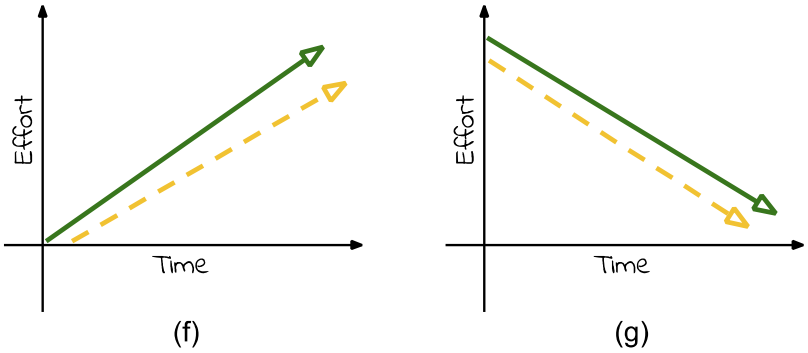

Figure 3: Allocation Process Patterns—Edge Cases
(f) Both architecture and feature work increase; (g) both types of work decrease.

4.7 Dependencies

The process patterns presented so far treat quality attributes and related architecture work as a single abstraction to be allocated with other work during software development. Here we switch focus to look at the individual quality attributes to examine the information needed to determine which QARs to work on at any given time and how to sequence them with respect to each other.

In the SQALE method, Letouzey proposes a dependency hierarchy for QARs that suggests a sequence of work for these approaches [Letouzey 2013].[2] Testability, reliability, and changeability are the qualities of highest priority, since without them a development team has no ability to work on others. (Note that testability is one form of measurement. It is not about the ability to test; it is about providing the data that shows that the quality attribute properties are achieved.)

In *Software by Numbers*, Denne and Cleland-Huang use dependencies from one quality attribute to another, and from quality attributes to minimally marketable features, to understand what needs to be worked on and when [Denne 2003]. Goal models such as those proposed by KAOS are another dependency mapping approach [Dardenne 2007].

[2] SQALE (Software Quality Assessment based on Lifecycle Expectations) is a method to support the evaluation of a software application source code. It is a generic method, independent of the language and tools for source code analysis.

In Figure 4, we can see how these concepts relate. We have three QARs in the backlog. We have three examples of how the QARs were allocated across the first three iterations of development. In the first example, we are primarily concerned with where and when our work should occur, that is, the allocation in which we can assign QARs to iterations without further refinement or decomposition. In the second example, we decompose S1 into constituent parts. Finally, in the third example, our concern is how to set measurable outcomes on our progress toward meeting S1, an illustration of ratcheting.

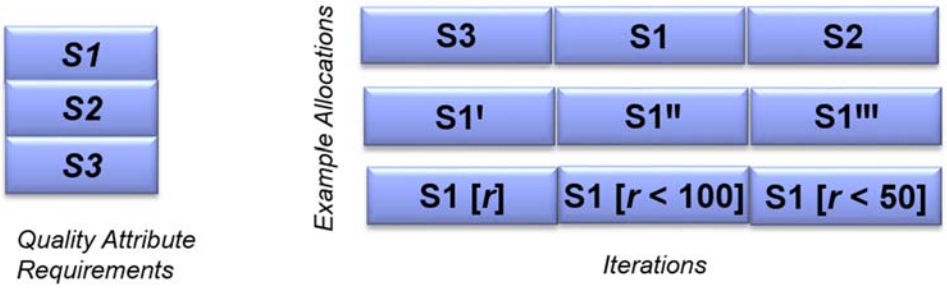

Figure 4: Allocating QARs Across Iterations of Development

Key Questions	*How does the development team properly value the nonfunctional stories?* *Is the story properly sized for work in one sprint?* *How constrained are the dependencies? Is it possible to work around a dependency?*

4.8 Empirical Evaluation of Allocation Approaches

We conducted a survey of three large organizations, two of which are Fortune 500 organizations. Figure 5 shows responses to a question that presented respondents with Figure 2 and asked them which pattern their most recent project best matched. Perhaps unsurprisingly, most projects did some form of up-front architecting. Next most common was parallel development, but a substantial number conducted agile development.

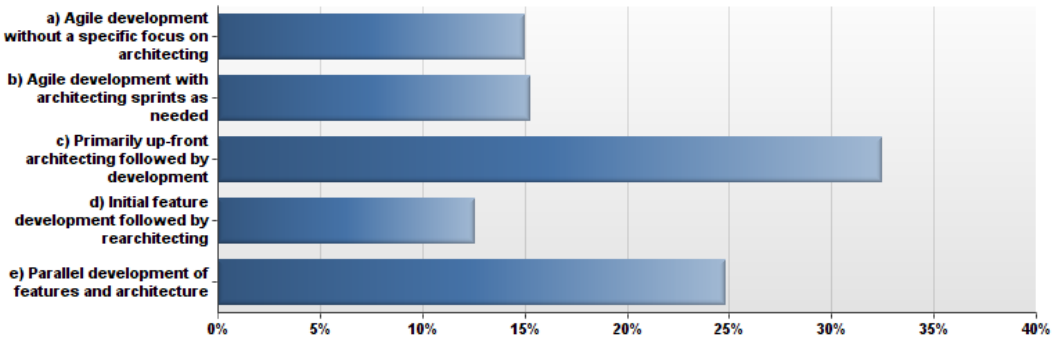

Figure 5: Survey Responses to Allocation Approaches
Letters match choices in Figure 2.

4.9 Summary

All of the patterns occur in practice, although some patterns occur more frequently in particular business contexts. For example, in a recent survey conducted at a large government contractor, we found that one-third of respondents were using up-front architecting (Pattern *c*), yet nearly as many were engaged in parallel architecting and development activities (Pattern *e*). Often a development team uses up-front architecting when there is a lengthy development life cycle, a safety-critical system context, or several regulatory bodies that must be involved in the development process.

The natural way these patterns are used is in combination. For example, it is not uncommon to see a development team start with evolving an existing system (Pattern *e*), followed by Iteration Zero to conduct some architecture planning to address new business goals (Pattern *c*), at which point the team puts more emphasis on feature development (Pattern *a*) and conducts no further architectural work until a need to introduce new QARs arises, when the effort switches to rework (Pattern *d*). Similarly, many successful agile software development projects start with an architecture runway (Pattern *e*) and supplement with a hardening sprint (Pattern *b*) [Bellomo 2013].

5 Using Existing Practices to Manage Iterations

We have shown that part of design in an iterative setting requires understanding how to refine and allocate QARs and functional requirements to iterations. This section offers an overview of how existing software analysis techniques and life-cycle planning methods can help a development team manage these iterations to enable incremental iterative development at larger scales. Here we summarize requirements elicitation and design-fragment analysis techniques including sprint planning, INVEST criteria, Planguage, set-based design, quality attribute elicitation models, and cost–benefit analysis methods. Then we review several software life-cycle methods including SAFe, disciplined agile delivery, and the incremental funding method. Using some combination of these techniques and methods, as appropriate to the software development effort, will help a development team employ architecture in service of smoothing the development of business capabilities.

5.1 Software Analysis Techniques

User Stories

The product backlog holds a collection of user stories, which are requirements expressed using (conventionally) the form "As a <user>, I would like to <activity>, in order to <goal>" [Cohn 2004]. The product owner and development team select stories according to immediate value. Backlog refinement is an ongoing process that ensures the stories in the backlog are appropriately refined and properly sized. Allocation happens immediately prior to the sprint. Wirfs-Brock uses "landing zones" to ensure that the development team has some flexibility in making tradeoffs among requirements [Wirfs-Brock 2011]. Each requirement has a range of acceptable values labeled *minimum*, *target*, or *outstanding*. A landing zone is similar to release criteria and allows for tolerances in acceptable values. Bjarnason and colleagues reported that for requirements, iterative development helped prevent overscoping [Bjarnason 2011].

INVEST Criteria

The INVEST criteria are that a requirement be "Independent, Negotiable, Valuable, Estimable, Small, and Testable" [Lawrence 2009]. These criteria are commonly used to evaluate the quality of a given user story, but clearly they map well to the notion of allocating and refining requirements. Independence, negotiability, and valuation are all linked to allocation, and estimableness and smallness are refinement guidelines. Testability is a value judgment to measure success. The criteria are subjective, so understanding definitions (e.g., how small?) in a given context is vital. However, it is not clear that these criteria are sufficient for good quality attribute requirements, as they were applied to functional user stories first.

Planguage

With Planguage, Gilb defines tags for architectural objectives (most often QARs) [Gilb 2007]. Planguage uses Scale, Meter, Minimum, Plan, and Wish tags to standardize stories. For example, for the response time component of a performance quality attribute, our scale is seconds, the meter is time between the user pressing a button and an outcome, the minimum acceptable outcome

is 7 seconds, the plan is for 4 seconds, and the wish is for 2 seconds. These tags provide a refinement approach to implementing the software. In other words, a development team can first ensure that the implementation will satisfy the minimum outcome, and in later iterations they can focus on improving response time to the plan or wish levels. Operationally, the team can use the Planguage technique to create independent requirements for allocation.

Set-Based Design

Traditionally cost estimation has involved fixed-point estimates: how long the project will take and how much it will cost. This is too absolute for most software projects, however, as noted by Kennedy and colleagues [Kennedy 2014]. They propose instead a set-based approach, in which the set defines a range of acceptable and likely outcomes. IBM's Walker Royce has espoused similar thinking [Royce 2011]. The developers define a response measure (useful for testability) for each quality attribute and then find the limit curve of the subcomponent of interest using prototype spikes. The goal is to learn from each iteration so that estimates become more and more tightly bound.

Quality Attribute Elicitation Models

Scenario-driven design refines high-level QARs into more specific scenarios. In a Quality Attribute Workshop, scenarios are the building blocks for eliciting feedback on a set of architectural strategies and business drivers [Barbacci 2003]. A scenario is described as specifically as necessary to exercise the desired quality attribute (for example, a login use case for the security attribute). Wood describes an approach that assigns scenarios to different iterations in the design phase of the system [Wood 2007]. The collection of individual scenarios is not exhaustive, however, and more coverage would be necessary to fully test all of the QARs.

Architectural tactics are design decisions that influence the achievement of a quality attribute response [Bass 2012]. A tactic tree, as proposed by Bass and colleagues, provides a rudimentary breakdown of such decisions for common quality attributes [Bass 2012]. These are likely approaches to decomposition. For example, for security we could detect attacks or resist attacks, and if we detect attacks we have a number of design choices, including detect intrusions, detect denial of service, and so on.

Patterns bundle design decisions (collections of tactics) into allocatable units to optimize the solution according to industry best practices for addressing common problems. Patterns may need to be broken down into smaller pieces to fit in an iteration, and tactics can give insight into their decomposition.

The technique described by Bass and colleagues captures QARs as six-part scenarios [Bass 2012]:

1. source of stimulus
2. stimulus
3. environment
4. artifact
5. response
6. response measure

This technique maps to the "Given/When/Then" behavior-driven development approach (also called "specification by example") [North 2006]. Parts 3 and 4 are the Given, Parts 1 and 2 the When, and Parts 5 and 6 the Then. If a quality attribute is amenable, one can add quality thresholds to existing stories or system tests and monitor outcomes (response measures). This serves as the *Testable* property. Value and cost remain outside the scope of a quality attribute scenario. This approach is particularly suited to runtime qualities such as availability and performance, which are easily operationalized. Design qualities such as maintainability, on the other hand, are not so simple to evaluate quantitatively (although tools are improving in this space). Leffingwell also notes that tests may vary from inspection, to special harnesses, to continuous monitoring [Leffingwell 2011a]. Key questions to investigate are whether quality attribute scenarios are elaborated at the right level of detail to allow a developer to complete them in a single iteration and how the QAR can be suitably operationalized for monitoring.

Cost–Benefit Analysis and Architecture Improvement

The Cost Benefit Analysis Method (CBAM) is a technique for evaluating architectural alternatives using stakeholder-driven utility curves [Kazman 2002]. This technique is primarily a means for understanding the cost and value of an architectural approach prior to undertaking it so that the development team works on the high-value activities. The utility curve shows how much more benefit moving from one threshold to another gives for how much cost. However, more research is needed to understand the incremental cost and rework associated with moving along the curve for given design fragments. One might also extend the concept of a utility curve to harness real data to provide iterative monitoring as the system is developed to support measurement-driven analysis. Related techniques in real-options analysis economically model the value of designs and search-based optimization of release plans [Sullivan 1999].

5.2 Software Life-Cycle Methodology

Scaled Agile Framework

Leffingwell's SAFe is a method for implementing iterative development in larger software projects (larger size usually means teams with more than 10 to 15 people collaborating on a single project) [Leffingwell 2014b]. There is well-developed guidance for moving from high-level "portfolio" projects to team-sized iteration elements. Requirements flow downstream to portfolio, product, and team backlogs. A special work stream handles architecture-related stories (the QARs). Architecture epics are ongoing cycles for refining and allocating architecture-related work to team-sized iterations [Leffingwell 2011b, 2014a]. A work-in-progress limit focuses the architecture team on a limited amount of work.

Disciplined Agile Delivery

Disciplined Agile Delivery [Ambler 2012] outlines an Inception Phase for software development. Similar to the Rational Unified Process, this practice's Iteration Zero is used for planning the iterations involved in the whole development life cycle. One scopes the project and identifies preliminary requirements, possible architectures, and unknowns for risk management and prototypes. As in SAFe, these high-level plans are then handed off to the iteration planning exercise. Throughout, the development team focuses on risk management to emphasize the value being delivered.

Incremental Funding Method

In the book *Software by Numbers*, Denne and Cleland-Huang introduce the Incremental Funding Method to manage software development projects [Denne 2003]. Their approach is primarily an economic one and centers on accurately assessing the net present value of software functionality. Functionality is refined into units of Minimally Marketable Features (MMFs), the smallest unit a customer would value. For each MMF, a refinement is proposed that highlights common architectural constraints for all MMFs. A dependency map then outlines the possible allocation patterns for building the MMF.

6 Related Approaches

There are several areas of research and practice that we do not probe in depth but that may have relevance, either in terms of formal tools for analysis or software tooling that could be leveraged for incremental iterative development at scale. These include slicing user stories, aspect-oriented software development, model slicing, valuation approaches, and static analysis.

First is the technique of slicing user stories in general, not only nonfunctional requirements. For example, as explained in Gottesdiener and Gorman, one can take a number of approaches to reduce any high-level user story, or *epic*, as it tends to be called [Gottesdiener 2010]. It can be broken into the six options for expansion mentioned in Section 3.4—user, actions, data, business rules, interfaces, and quality attributes—to create a much larger yet better described epic. All of the expansions form a possible option portfolio from which the most valuable slices can be chosen for immediate work.

Aspect-oriented software development (AOSD) allows quality attributes that are implemented as behavior, such as logging, to be woven into existing code [Kiczales 1997]. The idea is to support crosscutting concerns in the language itself. Each woven requirement could serve as a way to integrate slices into program development. Adoption of the AOSD tooling has been slow, but many properties of aspect-orientation might be found in dependency injection and configuration approaches for using software frameworks.

Model slicing, well described by Famelis and colleagues, applies formal logic to the problem of specifying a range of possibilities for a model-driven development [Famelis 2012]. The development team describes the system as a set of core objects and "possibilities," which represent trajectories that the system might take. Model slicing is similar to the way that agile methods provide support for uncertainty, only here the support is backed by a formalism that permits analysis of possible futures. Scalability in the face of the satisfiability problem's assumed intractability is a question, of course. Similar work exists in the requirements engineering community, both in search-based optimization [Zhang 2007] and in requirements "roadmaps." For example, Jureta and colleagues specify possible future requirements that might apply, to which the system can either self-adapt or that can provide a trajectory for the development effort (i.e., the roadmap specifies the tasks that ought to be undertaken) [Jureta 2010].

Underpinning the decomposition of a story is the need for prioritization or valuation approaches that assign a numeric value (possibly ordinal, possibly ratio). This might involve a simple weighting, like story points, or more complex economic approaches that use real-options analysis to assess the net present value of the quality attribute [Carriere 2010]. Economic models then allow for the introduction of industrial engineering theories for scheduling, such as weighted shortest job first [Reinertsen 2009]. Finally, there is a rich literature in program slicing related to static analysis. Here, the intent is to understand where and when a particular object of interest, typically a variable, is accessed or accessible. There is minimal overlap with story slicing, but some formal approaches may be useful. For example, if there were an algebra for describing architecturally significant requirements, one might apply slicing operators to refine that variable of interest. This is the approach underlying Khan et al. [Khan 2008].

7 Conclusion

This report has surveyed the current state of the art in refinement and allocation of QARs. A key finding is that the patterns of allocation we describe do not exist in isolation but in combinations. As a result, it is no surprise that no one technique is suitable to satisfy key QARs that are relevant to developing business capabilities. Development teams practice refinement in a number of ways, but ultimately the purpose is to decompose a possibly vague, nonuniform customer requirement or business goal into iteration-sized pieces. In the allocation process, developers then take those pieces and determine when, and why, to work on them. We characterized this process as a design activity: refinement and allocation are explorations of the problem and solution spaces, and evolutionary, iterative development allows for course changes when the development team acquires new information. Developers should work toward optimizing the satisfaction of QARs in the context of the cost of implementation, cost of rework, cost of delay, tradeoffs among multiple QARs, and ultimate value.

References

URLs are valid as of the publication date of this document.

[Adzic 2012]
Adzic, G. *Splitting User Stories: The Hamburger Method*. http://gojko.net/2012/01/23/splitting-user-stories-the-hamburger-method (2012).

[Ambler 2012]
Ambler, S. & Lines, M. *Disciplined Agile Delivery*. IBM Press, 2012.

[Anderson 2012]
Anderson, D. J. "Kanban at Scale and Why Traditional Approaches Set You Up for Failure." Presentation adapted from a keynote at OOP 2012. http://www.slideshare.net/agilemanager/kanban-largescalensn2012 (2012).

[Antón 1996]
Antón, A. I. "Goal-Based Requirements Analysis," 136–144. *Proceedings of the 2nd International Conference on Requirements Engineerin*g. Colorado Springs, CO, Apr. 1996. IEEE Computer Society Press, 1996.

[Apache 2014]
Apache Software Foundation. *Hadoop Distributed File System*. https://issues.apache.org/jira/browse/HDFS (2014).

[Bachmann 2003]
Bachmann, F.; Bass, L.; & Klein, M. H. "Moving from Quality Attribute Requirements to Architectural Decisions," 122–129. *Proceedings of the International SofTware Requirements to Architectures Workshop (STRAW'03)*. Portland, OR, May 2003. University of Waterloo, 2003.

[Barbacci 2003]
Barbacci, M. R.; Ellison, R. J.; Lattanze, A. J.; Stafford, J. A.; Weinstock, C. B.; & Wood, W. G. *Quality Attribute Workshops (QAWs), Third Edition* (CMU/SEI-2003-TR-016). Software Engineering Institute, Carnegie Mellon University, 2003. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=6687

[Basili 1975]
Basili, V. & Turner, A. J. "Iterative Enhancement: A Practical Technique for Software Development." *IEEE Transactions on Software Engineering 1*, 4 (1975): 390–296.

[Bass 2012]
Bass, L.; Clements, P.; & Kazman, R. *Software Architecture in Practice*, 3rd ed. Addison-Wesley Professional, 2012.

[Beck 2001]
Beck, K. et al. *Agile Manifesto*. http://agilemanifesto.org (2001).

[Bellomo 2013]
Bellomo, S.; Nord, R. L.; & Ozkaya, I. “A Study of Enabling Factors for Rapid Fielding: Combined Practices to Balance Tension Between Speed and Stability,” 982–991. *Proceedings of the 2013 International Conference on Software Engineering*. San Francisco, CA, May 2013. IEEE Computer Society Press, 2013.

[Bellomo 2014a]
Bellomo, S.; Kruchten, P.; Nord, R. L.; & Ozkaya, I. “How to Agilely Architect an Agile Architecture.” *Cutter IT Journal 27*, 2 (2014): 12–17.

[Bellomo 2014b]
Bellomo, S.; Ernst, N.; Nord, R. L.; & Ozkaya, I. “Evolutionary Improvements of Cross-Cutting Concerns: Performance in Practice,” 545–548. *International Conference on Software Maintenance and Evolution*. Victoria, BC, Oct. 2014. IEEE Computer Society Press, 2014.

[Bjarnason 2011]
Bjarnason, E.; Wnuk, K.; & Regnell, B. “A Case Study on Benefits and Side-Effects of Agile Practices in Large-Scale Requirements Engineering,” Article 3. *Proceedings of the 1st Workshop on Agile Requirements Engineering at the European Conference on Object-Oriented Programming*. Lancaster, UK, July 2011. ACM, 2011.

[Boehm 2003]
Boehm, B. *Balancing Agility and Discipline: A Guide for the Perplexed*. Addison-Wesley, 2003.

[Breivold 2012]
Breivold, H. P.; Crnkovic, I.; & Larsson, M. “A Systematic Review of Software Architecture Evolution Research.” *Information & Software Technology 54*, 1 (2012): 16–40.

[Brown 2011]
Brown, N..; Nord, R. L.; Ozkaya, I.; & Pais, M. “Analysis and Management of Architectural Dependencies in Iterative Release Planning,” 103–112. *Proceedings of the 9th Working IEEE/IFIP Conference on Software Architecture*. Boulder, CO, June 2011. IEEE Computer Society Press, 2011.

[Cao 2008]
Cao, L. & Ramesh, B. “Agile Requirements Engineering Practices: An Empirical Study.” *IEEE Software 25*, 1 (Jan. 2008): 60–67.

[Carriere 2010]
Carriere, J.; Kazman, R.; & Ozkaya, I. “A Cost-Benefit Framework for Making Architectural Decisions in a Business Context,” 149–157. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*. Cape Town, South Africa, May 2010. ACM, 2010.

[Cervantes 2013]
Cervantes, H.; Velasco-Elizondo, P.; & Kazman, R. “A Principled Way to Use Frameworks in Architecture Design.” *IEEE Software 30*, 2 (2013): 46–53.

[Cockburn 1996]
Cockburn, A. *Walking Skeleton*. http://alistair.cockburn.us/Walking+skeleton (1996).

[Cockburn 2008]
Cockburn, A. *Elephant Carpaccio*. http://alistair.cockburn.us/Elephant+carpaccio (2008).

[Cohn 2004]
Cohn, M. "User Stories Applied." Addison-Wesley Professional, 2004.

[CONNECT 2013]
CONNECT. *Sprint 120 Progress Summary*. https://connectopensource.atlassian.net/wiki/display/NHINProgress/Sprint+120+Progress+Summary (2013).

[CONNECT 2014]
CONNECT. *Requirement Artifacts*. https://connectopensource.atlassian.net/wiki/display/CONNECTWIKI/Requirements+Artifacts (2014).

[Dardenne 2007]
Dardenne, A.; van Lamsweerde, A.; & Fickas, S. "Goal-Directed Requirements Acquisition." *Science of Computer Programming 20*, 1/2 (Nov. 2007): 1–36.

[Denne 2003]
Denne, M. & Cleland-Huang, J. *Software by Numbers*. Prentice Hall, 2003.

[Ernst 2014]
Ernst, N. & Bellomo, S. *Evolutionary Improvements of Quality Attributes: Performance in Practice*. Software Engineering Institute, Carnegie Mellon University, Sep. 2014. http://blog.sei.cmu.edu/post.cfm/evolutionary-improvements-quality-attributes-251

[Famelis 2012]
Famelis, M.; Salay, R.; & Chechik, M. "Partial Models: Towards Modeling and Reasoning with Uncertainty," 573–583. *Proceedings of the 34th International Conference on Software Engineering*. Zurich, Switzerland, June 2012. IEEE Computer Society Press, 2012.

[Galen 2014]
Galen, R. "Hardening Sprints: The Good, Bad, and Downright Ugly." *Agile Record*. http://www.agilerecord.com/hardening-sprints (2014).

[Gilb 2007]
Gilb, K. *Evo: Evolutionary Project Management & Product Development*. http://gilb.com/tiki-download_file.php?fileId=27 (2007).

[Gottesdiener 2010]
Gottesdiener, E. & Gorman, M. "Slicing Requirements for Agile Success." *Better Software Magazine* (July/Aug. 2010): 16–21.

[Green 2013]
Green, P. *Splitting Stories into Small, Vertical Slices*. Agile @ Adobe, 2013. http://blogs.adobe.com/agile/2013/09/27/splitting-stories-into-small-vertical-slices

[Humble 2010]
Humble, J. & Farley, D. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Pearson Education, 2010.

[Hunt 1999]
Hunt, A. & Thomas, D. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Professional, 1999.

[Jureta 2010]
Jureta, I. J.; Borgida, A.; Ernst, N.; & Mylopoulos, J. "Techne: Towards a New Generation of Requirements Modeling Languages with Goals, Preferences, and Inconsistency Handling," 115–124. *Proceedings of the 18th International Conference on Requirements Engineering*. Sydney, Australia, Sep. 2010. IEEE Computer Society Press, 2010.

[Kazman 2002]
Kazman, R.; Asundi, J.; & Klein, M. H. *Making Architecture Design Decisions: An Economic Approach* (CMU/SEI-2002-TR-035). Software Engineering Institute, Carnegie Mellon University, 2002. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=6265

[Kennedy 2014]
Kennedy, B. M.; Sobek, II, D. K.; & Kennedy, M. N. "Reducing Rework by Applying Set-Based Practices Early in the Systems Engineering Process." *Systems Engineering 17*, 3 (2014): 278–296.

[Khan 2008]
Khan, S. S., Greenwood, P., Garcia, A., Rashid, A. "On the Impact of Evolving Requirements-Architecture Dependencies: An Exploratory Study," 243–257. *Proceedings of the International Conference on Advanced Information Systems Engineering*. Montpellier, France, June 2008. Springer, 2008. http://dx.doi.org/10.1007/978-3-540-69534-9_19

[Khan 2014]
@aslamkhn (Aslam Khan). *I'm going to say it again….* Twitter. 12:02 AM, June 11, 2014. https://twitter.com/aslamkhn/status/476620594507939840

[Kiczales 1997]
Kiczales, G.; Lamping, J.; Menhdhekar, A.; Lopes, C. V.; Maeda, C.; Loingtier, J.-M.; & Irwin, J. "Aspect-Oriented Programming," 220–242. *Proceedings of the European Conference on Object-Oriented Programming*. Jyväskylä, Finland, June 1997. Springer, 1997.

[Kruchten 1998]
Kruchten, P. *The Rational Unified Process: An Introduction*. Addison-Wesley, 1998.

[Kua 2013]
Kua, P. *An Appropriate Use of Metrics*. Martin Fowler, Feb. 2013. http://martinfowler.com/articles/useOfMetrics.html

[Lawrence 2009]
Lawrence, R. *Patterns for Splitting User Stories*. Agile for All, Oct. 2009. http://www.agile-forall.com/2009/10/patterns-for-splitting-user-stories

[Leffingwell 2008]
Leffingwell, D.; Martens, R.; & Zamora, M. *Principles of Agile Architecture: Intentional Architecture in Enterprise-Class Systems*. Scaling Software Agility, Aug. 2008. http://scalingsoftwareagilityblog.com/wp-content/uploads/2008/08/principles_agile_architecture.pdf

[Leffingwell 2011a]
Leffingwell, D. *Agile Software Development with Verification and Validation in High Assurance and Regulated Environments* (Whitepaper). Rally Software, 2011. https://www.rallydev.com/resource/agile-software-development-verification-and-validation-high-assurance-and-regulated-0

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

[Leffingwell 2014a]
Leffingwell, D. *Architectural Epic Kanban*. http://scaledagileframework.com/architectural-epic-kanban (2014).

[Leffingwell 2014b]
Leffingwell, D. *Scaled Agile Framework (SAFe)*. http://scaledagileframework.com (2011–2014).

[Letouzey 2013]
Letouzey, J.-L. & Ilkiewicz, M. "Managing Technical Debt with the SQALE Method." *IEEE Software 29*, 6 (2013): 44–51.

[Nord 2012]
Nord, R. L.; Ozkaya, I.; Kruchten, P.; & Gonzalez-Rojas, M. "In Search of a Metric for Managing Architectural Technical Debt," 91–100. *Proceedings of the Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture (ECSA)*. Helsinki, Finland, Aug. 2012. IEEE Computer Society Press, 2012.

[Nord 2014]
Nord, R. L.; Ozkaya, I.; Sangwan, R. S.; & Koontz, R. J. "Architectural Dependency Analysis to Understand Rework Costs for Safety-Critical Systems," 185–194. *ICSE Companion 2014: Companion Proceedings of the 36th International Conference on Software Engineering*. Hyderabad, India, July 2014. ACM, 2014.

[North 2006]
North, D. "Behavior Modification: The Evolution of Behavior-Driven Development." *Better Software Magazine* (June 2006). http://www.stickyminds.com/better-software-magazine/behavior-modification

[Nuseibeh 2001]
Nuseibeh, B. A. "Weaving Together Requirements and Architectures." *Computer 34*, 3 (2001): 115–117.

[Reinertsen 2009]
Reinertsen, D. G. *The Principles of Product Development Flow: Second Generation Lean Product Development*. Celeritas, 2009.

[Royce 2011]
Royce, W. "Measuring Agility and Architectural Integrity." *International Journal of Software and Informatics 5*, 3 (2011): 415–433.

[Séguin 2012]
Séguin, N.; Tremblay, G.; & Bagane, H. "Agile Principles as Software Engineering Principles: An Analysis," 1–15. *Lecture Notes in Business Information Processing*, Vol. 111. Edited by C. Wohlin. Springer, 2012.

[Sethi 2009]
Sethi, K.; Cai, Y.; Wong, S.; Garcia, A.; & Sant'Anna, C. "From Retrospect to Prospect: Assessing Modularity and Stability from Software Architecture," 269–272. *Proceedings of the IEEE/IFIP Working International Conference on Software Architecture (WICSA'09)*. Cambridge, U.K., Sep. 2009. IEEE Computer Society Press, 2009.

[Sullivan 1999]
Sullivan, K.; Chalasani, P.; & Jha, S. "Software Design as an Investment Activity: A Real Options Perspective," 215–262. *Real Options and Business Strategy: Applications to Decision Making*. Edited by K. Sullivan, P. Chalasani, & S. Jha. Risk Books, 1999.

[Verwijs 2013]
Verwijs, C. *8 Useful Strategies for Splitting Large User Stories*. http://www.christiaanverwijs.nl/post/2013/05/17/8-useful-strategies-for-splitting-large-user-stories-%28and-a-cheat-sheet%29.aspx (2013).

[Wirfs-Brock 2011]
Wirfs-Brock, R. *Introducing Landing Zones*. http://wirfs-brock.com/blog/2011/07/20/introducing-landing-zones (2011).

[Wood 2007]
Wood, W. *A Practical Example of Applying Attribute-Driven Design (ADD), Version 2.0* (CMU/SEI-2007-TR-005). Software Engineering Institute, Carnegie Mellon University, 2007. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=8319

[Zhang 2007]
Zhang, Y.; Harman, M.; & Mansouri, S. A. "The Multi-Objective Next Release Problem," 1129–1137. *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*. London, July 2007. ACM, 2007.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	June 2015	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Enabling Incremental Iterative Development at Scale: Quality Attribute Refinement and Allocation in Practice	FA8721-05-C-0003

6. AUTHOR(S)

Neil Ernst, Stephany Bellomo, Robert L. Nord, and Ipek Ozkaya

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2015-TR-008

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFLCMC/PZE/Hanscom Enterprise Acquisition Division 20 Schilling Circle Building 1305 Hanscom AFB, MA 01731-2116	

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

Lengthy requirements, design, integration, test, and assurance cycles delay delivery, resulting in late discovery of mismatched assumptions and system-level rework. In response, development methods that enable frequent iterations with small increments of functionality, such as agile practices, have become popular. But such methods de-emphasize architectural analysis; they assume the emergence or existence of a stable architecture. Yet as the business goals and context evolve, the architecture must also change, which requires allocating increments of quality attribute requirements to iterations along with other business capabilities. Quality attribute requirements (also called nonfunctional requirements) are hard to separate into smaller increments since they often crosscut many aspects of the product. As a result, allocation is uneven since it is challenging to decompose them and understand their value. Working with quality attribute requirements in an incremental and iterative fashion involves solving two problems: separating high-level requirements into their constituent parts and allocating them to iterations to fulfill the requirement. Underpinning both problems is the need for measurements to show that the requirement is satisfied. This report describes industry principles and practices used to smooth the development of business capabilities and suggests some approaches to enabling large-scale iterative development, or “agile at scale.”

14. SUBJECT TERMS	15. NUMBER OF PAGES
agile at scale, architectural analysis, incremental development, large-scale development, quality attributes, requirements	35

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL