Chapter 1

A Brief History of Software Requirements Methods

Inertia is the residue of past innovation efforts. Left unmanaged, it consumes the resources required to fund next generation innovation.

—Geoffrey Moore, *Dealing with Darwin*

Software Requirements in Context: Decades of Advancing Software Process Models

Software development has become one of the world's most important technologies. The software we produce today is rapidly becoming the embodiment of much of the world's intellectual property. Simply put, our modern world depends on software.

In support of our efforts over the past 40 to 50 years, we have implemented various software development methodologies—process frameworks we use to structure, manage, and control our work. Early on, it was a "cut-and-try" approach—ad hoc—as and where necessary. In large part, that worked.

Over time, the scope and reach of our endeavors, along with the power of the computers we programmed, increased by 10,000 fold. It seems like very quickly we went from simple simulations to flying commercial airliners internationally. So, the consequences of success or failure—whether measured in potential economic or human cost—increased exponentially as well. To mitigate all this new risk and help us produce only intended, rather than unintended, outcomes, we developed more structured and controlled methodologies.

Because what we produce is not physical goods but intangible ideas reflected in binary code, our methods all had a primary focus on "managing software requirements." *Software requirements* was the label we applied to the abstractions we use to carry the value stream into development and on to delivery to our customers. In large part, these newer practices worked too, and we owe much of our successes to these methods. Indeed, we shipped a *lot* of software.

But over time, the applications we developed became larger and larger still, and the methods we used to control our work became heavier and heavier. As an unintended

consequence, we started slowing down the very thing we were trying to speed up—our ability to deliver higher-value, higher-quality software at faster and faster rates.

So, in the past two decades, the movement to more "agile" and "leaner" software development methodologies, including lighter weight but still safe and effective treatment of application requirements, has been one of the most significant factors affecting the industry. Simply, we need processes that provide even better safety and governance than we have experienced but without the burden. We want the best of both worlds.

So, this migration to more rapidly exploratory and lighter-weight processes has been a consistent theme over time, as Figure 1–1 shows.

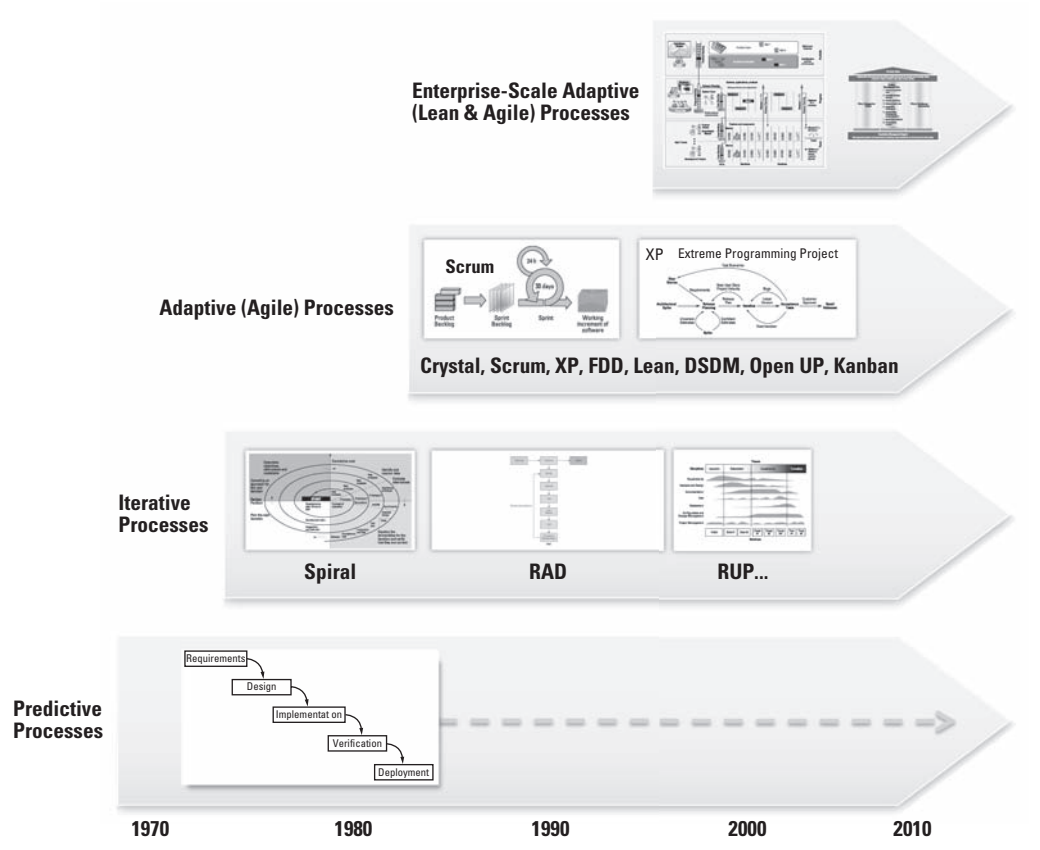


Figure 1–1 Software process movements over the past few decades
Adapted from Trail Ridge Consulting, LLC

A brief look at each of these mega-software process trends will tell us where we've been, where we are today, and perhaps a bit about where we are likely headed in the future.

PREDICTIVE, WATERFALL-LIKE PROCESSES

The software industry advanced quickly after its inception in the 1950s and 1960s. As it did so, the need to be able to better predict and control ever larger-scale software project outcomes somehow led us to what has become known as the sequential, stage-gated "waterfall" software process model, usually typified by a graphic such as Figure 1–2.

In this model, software development occurred in an orderly series of sequential stages. Requirements were agreed to, a design was created, and code followed thereafter. Lastly, the software was tested to verify its conformance to its requirements and design.

Winston Royce, who was at TRW at the time, is often credited with the creation of this model. Ironically, however, Royce actually described it as a model that could *not* be recommended for large-scale software development. In his seminal paper [Royce 1970], he wrote this:

> In my experience, the simpler model . . . [such as the one pictured in Figure 1–2] has never worked on large software development efforts.

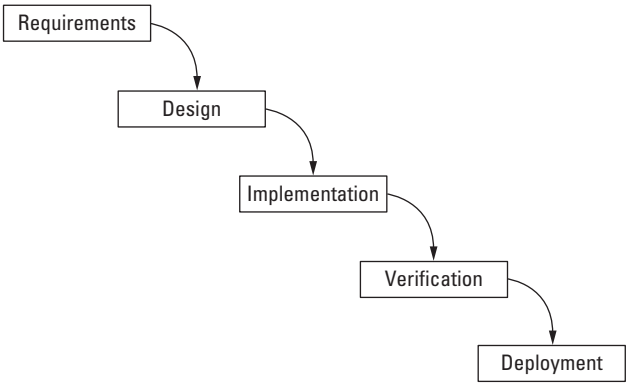


Figure 1–2 Simplified "waterfall" model. Progress flows top to bottom, like a waterfall.

He then went on to describe an enhanced model, which included building a prototype first and then using the prototype plus feedback between phases to build a final deployment. Unfortunately, his actual guidance is lost to history, or perhaps to the beguiling, construction-like thinking and oversimplification of the simple graphic in Figure 1–2. So, what came into common usage was not what Royce intended.

Problems with the Model

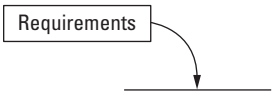
As Royce would have likely predicted, the model hasn't worked all that well for large software projects, and we have all struggled under the burden. Within just a decade or two, the resulting statistics were not very pretty. For example, the oft-cited Standish Group's Chaos report survey [Standish 1994[1]] noted the following.

- 31% of projects will be canceled before they are completed.
- 53% of the projects will cost more than 189% of their estimates.
- Only 16% of projects were completed on time and on budget.
- For the largest companies, completed projects delivered only 42% of the original features and functions.

In addition, it appears that an ineffective treatment of requirements was a primary root cause, because these were the three most common factors that caused projects to be "challenged":

- *Lack of user input:* 13% of all projects
- *Incomplete requirements and specifications:* 12% of all projects
- *Changing requirements and specifications:* 12% percent of all projects

Requirements in the Waterfall Model: The Iron Triangle



Of course, we can't say for certain that applying a rigid waterfall-based requirements process was the root cause, but in any case it's clear that misunderstood and changing requirements had a huge impact on project success, and the waterfall model was the dominant process model in effect at the time. Moreover, as generally applied, the deceptively simply "requirements box" pictured in the waterfall model implies that there is a set of requirements that can be reasonably determined "up front" and that these can then be used as a basis to estimate the schedule and budget of the project, as Figure 1–3 shows.

1. This often-cited report does have its critics. See *www.few.vu.nl/~x/chaos/chaos.pdf*. However, the general conclusions correlate pretty well to the author's experiences.

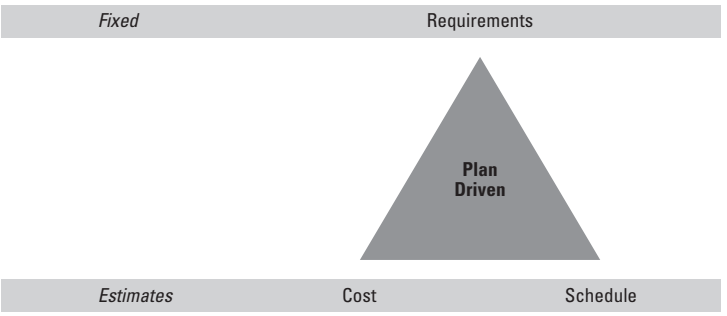


Figure 1–3 Once requirements are "known," you can estimate the cost and schedule.

Of course, from that point forward, the schedule and budget were likely also fixed (after all, what business could possibly plan for highly variable resources and project costing?). So, from a more realistic perspective, that led us to a simple "iron triangle" trap, as Figure 1–4 illustrates.

This "fixed requirements scope" assumption has indeed been found to be a root cause of project failure. For example, one key study of 1,027 IT projects in the United Kingdom [Thomas 2001] reported this: "Scope management related to attempting waterfall practices was the single largest contributing factor for failure." Here's the study's conclusion:

> This suggests that . . . the approach of full requirements definition, followed by a long gap before those requirements are delivered, is no longer appropriate. The high ranking of changing business requirements suggests that any assumption that there will be little significant change to requirements once they have been documented is fundamentally flawed.

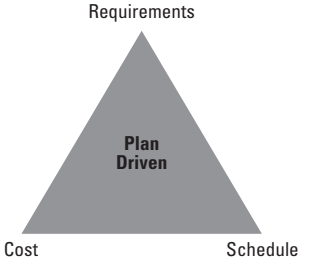


Figure 1–4 The iron triangle trap of the waterfall model

And What About Quality?

There is another critical, implicit deficit of the model as well. Software veterans reading this book will likely note that there is another critical dimension, *quality*, which does not even appear in the iron triangle diagram. Since cost, schedule, and requirements were *all* fixed, quality was the only *variable* the teams could access. We all know how well that worked out.

So, it certainly does appear from the data that attempting to *fix the requirements up front* and then *carefully control change* and *use quality as the team's variable* was a fundamentally flawed set of assumptions on which to base a software process model. This kind of hard data, plus our aggregate personal experiences, led us to a "tipping point" that warranted serious consideration of substantially different process models for the industry, and indeed they did evolve in due course.

And Yet, the Waterfall Model Is Still Amongst Us

Even in light of this evidence, however, the waterfall model is *still* in widespread use today, as Figure 1–5 implies.

Given its deficiencies, one wonders why that may be the case in 2010 and beyond. Perhaps there are a number of understandable reasons.

- The model was itself born as a fix to an earlier problem, which was the "code it, fix-it, code-it-some-more-until-it's-quickly-not-maintainable" tendency of prior practices.
- It appears to be extremely logical and prescriptive. Understand requirements. Design a system that conforms. Code it. Test it. What could be more sensible and logical than that?
- It worked to a point (we did and still do ship a *lot* of software using the model). As a result, companies built their project and program governance models, including business case and investment approvals, project review and quality assurance milestones, and the like, around its flawed software life cycle.
- It reflects a continuing market reality—customers still do impose fixed-date/fixed-requirements agreements on suppliers, and they will likely continue to do so for years to come. (And, yes, sometimes we impose them on ourselves.)

So, we belabor it here, not to further "beat a dead horse" but to recognize that this particular horse is likely to be still alive and kicking in many business contexts. No matter how agile we want to be, we will have to avoid its flailing hooves well into the future. In turn, this can severely impact our operating freedom in implementing agile requirements practices. The bigger the opportunity for gain in the larger setting, the more likely the old model still exists—and the bigger the obstacles we are likely to encounter!

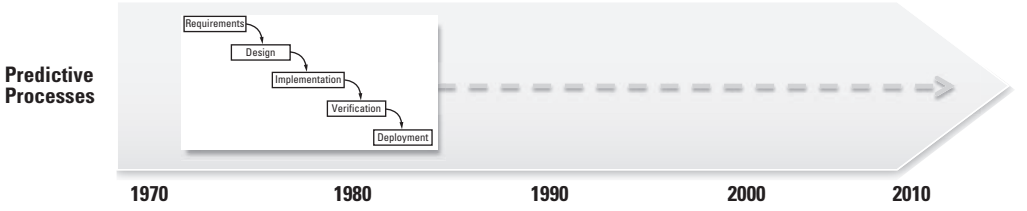


Figure 1–5 The model may have originated in the 1970s, but it is still in use today.

As agilists, our job is to help the business migrate to the new agile paradigm as efficiently as possible. Therefore, although we can agree that we don't want to support the waterfall model any longer than we absolutely have to, *we do have to understand it* and recognize that it still exists.

ITERATIVE AND INCREMENTAL PROCESSES

In the decades that followed, failures of the waterfall model, along with increasing time-to-market pressures and advances in software development tools and technologies, drove the need for more innovative, *discovery-based* models, which led us to the iterative processes of the 1980s and 1990s, as illustrated in Figure 1–6.

Generally, these can be seen as a continuum of increasingly iterative methods that used the following:

- Rapid development of understanding via experimental discovery (spiral)
- Rapid build of models, prototypes, and initial systems using more advanced tools (RAD)
- Iterative and incremental development of ever larger and more complex systems (RUP)

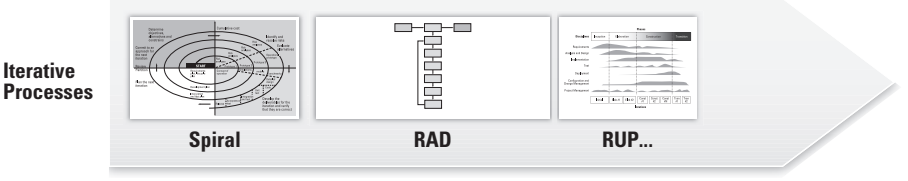
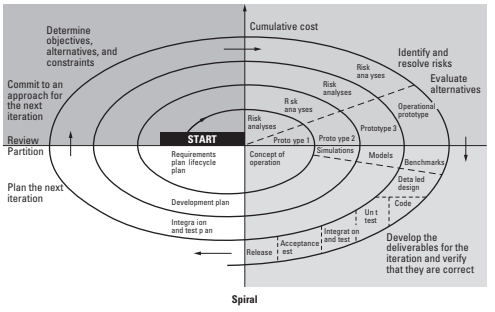


Figure 1–6 Iterative processes: spiral, RAD, and RUP

Spiral Model

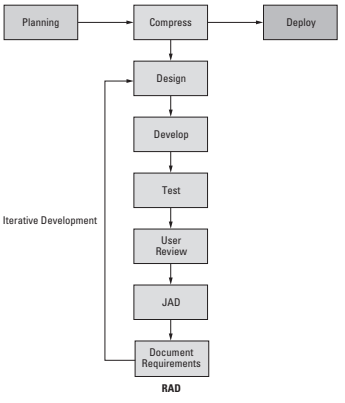


Source: B.W. Boehm, "A Spiral Model of Software Development and Enhancement," IEEE Computer, Vol. 21, No. 5, 1988, pp. 61–72

One trendsetter in this continuum [Boehm 1988] is widely credited for initiating this trend with what has been described as the "spiral model of development."

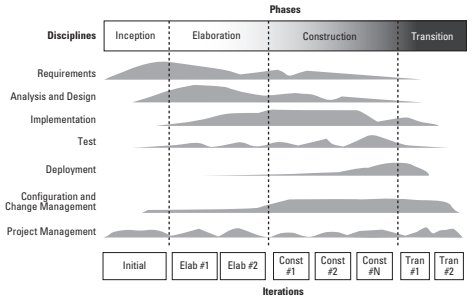
In the spiral model, requirements still have a strong early placeholder. An initial pass around the spiral is intended primarily to understand requirements and perform some validation of the requirements before more serious development begins. Thereafter, the model assumed another, larger "spiral" intended to develop the solution in largely sequential steps of design, coding, integration, and testing. As such, this was the first published model based on a more discovery-based requirements process (albeit followed by a fairly traditional sequential, waterfall-like process, but one that incorporated constant feedback). Most give credit to this thinking as the starting point to the iterative and incremental (and now, increasingly agile) software development methods movement.

Rapid Application Development



Rapid Application Development (RAD) is now considered to be a more generic term for a type of development process introduced by James Martin in the early 1990s. Although it was originally a fairly well-defined software process model focusing on the iterative development and construction of an increasingly capable series of prototypes, today it generally stands for any number of lighter-weight approaches, using fourth-generation languages and frameworks (such as web application frameworks), which accelerate the availability of working software. These frameworks and higher-level approaches often emphasize speed of deployment and feedback over performance and scalability. From a requirements perspective, the assumption was that if you could build it fast enough before the requirements changed, you would be more successful. And if you did get it wrong, the tools are sufficiently facile and lightweight that you could build it again faster than you could use traditional, paper-based requirements discovery methods. This thinking was indeed a harbinger of methods to come.

Rational Unified Process



Source: Leffingwell, *Scaling Software Agility: Best Practices for Large Enterprises*, Figure "RUP," © 2007 Pearson Education, Inc. Reproduced by permission of Pearson Education, Inc.

Based more on the spiral model than RAD and intended for large-scale applications where robustness, scalability, and extensibility were mandatory, the Rational Unified Process (RUP) was launched in the late 1990s. RUP is a widely adopted iterative and incremental software process model, actively marketed and supported by the Rational Software Division of IBM. RUP has proven to be an effective framework for the practice guidance and management of large-scale application development. It has seen widespread industry use (more than a million practitioners) and has been applied with success on thousands of projects of all types, including projects at the very largest scale.

RUP was the first widely adopted software process that recognized the necessary overlap of the various *activities* that occurred during the life cycle *phases* of *inception*, *elaboration*, *construction*, and *transition*. For example, activities such as "requirements" were no longer relegated to a single phase. Although requirements activities were particularly intensive during the early inception and elaboration phases (as illustrated by the size of the "humps" in the diagram), requirements elaboration and requirements change are considered to be a continuous process that occurs throughout the life cycle.

More recently, a number of lighter-weight and more agile instantiations of RUP, including Agile RUP and OpenUP (an open source process under the auspices of the Eclipse foundation[2]), have become available.

Requirements in Iterative Processes

In iterative processes, we see a purposeful move away from the traditional *big, up-front design* (BUFD) requirements and design artifacts, such as software requirements specifications, design specifications, and the like, which served to define and govern implementations in waterfall implementations. In its place, we see a "discovery-based" approach. In the iterative model, we applied lighter-weight documents and models such as vision documents, use-case models, and so on, which are used to initially define what is to be built. Based on these initial understandings, the iterative process itself is then applied to more quickly discover the "real user requirements" in early iterations, thus substantially reducing the overall risk profile of the project.

2. *www.eclipse.org/epf/*

Once better defined in early iterations, these requirements were then implemented in a fairly robust but mostly traditional build-out of code, tests, and so on, to implement the requirements and provide assurances that the system conformed to the agreed-to behaviors.

Clearly, this was a giant step forward for the industry and one that started to soften the boundaries of the iron triangle.

ADAPTIVE (AGILE) PROCESSES

Starting in the late 1990s and through the current decade, software process has seen an explosion of lighter-weight and ever-more-adaptive models. Based on some fundamental changing software implementation paradigms such as object orientation, 3G languages, and test-driven development, these models were based on a different economic foundation. These models assumed that—with the right development tools and practices—it was simply more cost effective to write the code quickly, have it evaluated by customers in actual use, be "wrong" (if necessary), and quickly refactor it than it was to try to anticipate and document all the requirements up front.

Indeed, the number of methods—including Dynamic Systems Development Method (DSDM), Feature-Driven Development (FDD), Adaptive Software Development, Scrum, Extreme Programming (XP), Open Unified Process (Open UP), Agile RUP, Kanban, Lean, Crystal Methods, and so on—speaks to the industry's thirst and constant drive for more effective and lighter-weight processes.

The Agile Manifesto

In 2001, the creators of many of the agile software development methodologies came together with others who were also implementing various agile methods in the field and created an Agile Manifesto[3] summarizing their belief that there is a better way to produce software. Even today it does an excellent job of synthesizing and defining the core beliefs underlying the movement:

> *We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:*
>
> | | | |
> |---|---|---|
> | Individuals and interactions | over | processes and tools |
> | Customer collaboration | over | contract negotiation |

3. *www.agilemanifesto.org*

Working software	over	comprehensive documentation
Responding to change	over	following a plan

That is, while there is value in the items on the right, we value the items on the left more.

Behind the manifesto itself are a set of core principles that serve as a common framework for all agile methods.

- *Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.*
- *Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.*
- *Working software is the primary measure of progress.*
- *Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.*
- *Business people and developers must work together daily throughout the project.*
- *Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.*
- *The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.*
- *Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.*
- *Continuous attention to technical excellence and good design enhances agility.*
- *Simplicity—the art of maximizing the amount of work not done—is essential.*
- *The best architectures, requirements, and designs emerge from self-organizing teams.*
- *At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.*

Given the number of agile methods, a reasonable treatment of each is outside our scope. However, according to a recent survey, it looks like these "method wars" have settled a bit, at least with respect to market share (for now), as Figure 1–7 shows.[4]

4. *www.versionone.com*

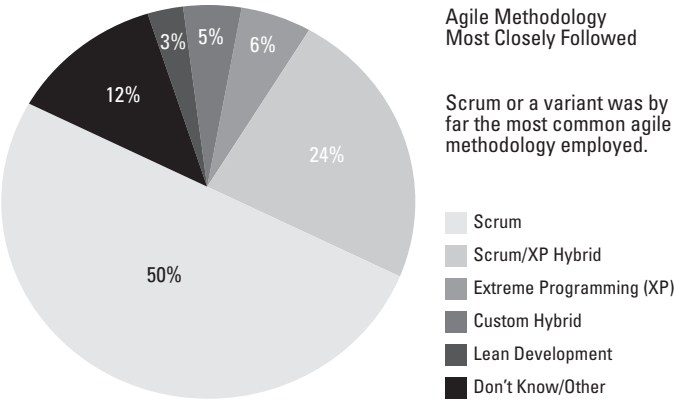
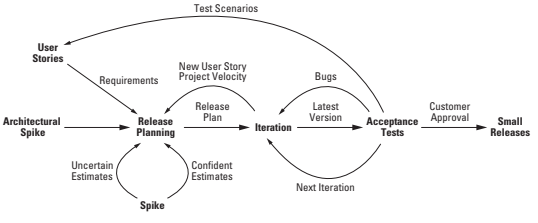


Figure 1–7 Survey of most widely adopted agile methods. Fourth Annual State of Agile Development Survey 2009. Courtesy of VersionOne, Inc.
Source: VersionOne's 2009 Agile Methodology Survey

The survey reflects that, currently, the most widely adopted agile methods are Scrum and XP. According to this survey, Scrum (with or without combination with XP) is now applied in 74% of agile implementations, and this has been our experience as well.

Based on their predominance, we'll be using these two methods as base practices for much of what we discuss in our agile requirements practices in this book, so a brief description of these is in order.

Extreme Programming (XP)



Source: Leffingwell, *Scaling Software Agility: Best Practices for Large Enterprises*, Figure "Extreme Programming Project" © 2007 Pearson Education, Inc. Reproduced by permission of Pearson Education, Inc.

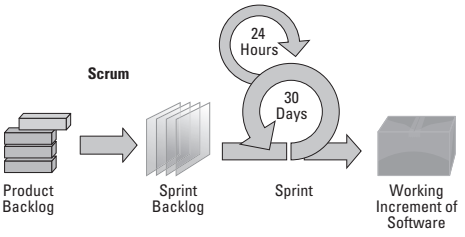
XP is a widely used agile software development method that is described in a number of books by Beck and others [Beck 2000; Beck and Andres 2005]. Key practices of XP include the following.

- A team of five to ten programmers work at one location with customer representation on-site.
- Development occurs in frequent builds or iterations, which may or may not be releasable, and delivers incremental functionality.
- Requirements are specified as user stories, each a chunk of new functionality the user requires.

- Programmers work in pairs, follow strict coding standards, and do their own unit testing. Customers participate in acceptance testing.
- Requirements, architecture, and design emerge over the course of the project.

XP is prescriptive in scope and is typically applied in small teams of less than ten developers, where the customer is integral to the team or readily accessible. In addition, the *P* in XP stands for *programming*, and as opposed to other methods, XP describes some strict practices for coding that have been shown to produce extremely high-quality output.

Scrum



Scrum is an agile project management method [Schwaber 2004] that is enjoying increasing widespread use. Key Scrum practices include the following.

- Work is done in "sprints," which are timeboxed iterations of a fixed 30 days or fewer duration.
- Work within a sprint is fixed. Once the scope of a sprint is committed, no additional functionality can be added, except by the development team.
- All work to be done is characterized as product backlog, which includes new requirements to be delivered, the defect workload, and infrastructure and design activities.
- A *Scrum Master* mentors the empowered, self-organizing, and self-accountable teams that are responsible for delivery of successful outcomes at each sprint.
- A *product owner* plays the role of the customer proxy.
- A daily stand-up meeting is a primary communication method.
- A heavy focus is placed on timeboxing. Sprints, stand-up meetings, release review meetings, and the like are all completed in prescribed times.
- Typical Scrum guidance calls for fixed 30-day sprints, with approximately 3 sprints per release, thus supporting incremental market releases on a 90-day time frame.

Scrum is achieving widespread use because it is a lightweight framework, and—more importantly—it works. It also has the added benefit of a training certification process, administered by the Scrum Alliance,[5] which is also a good source for ongoing discussions about the Scrum method, its application, and adoption.

5. *www.scrumalliance.org*

REQUIREMENTS MANAGEMENT IN AGILE IS FUNDAMENTALLY DIFFERENT

No matter the specific method, agile's treatment of requirements is fundamentally different. We see it immediately in the core principles:

> *Manifesto principle #1—Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.*
>
> *Manifesto principle #2—Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.*

Overall, the impact on the industry is dramatic and material. As described in *Scaling Software Agility* [Leffingwell 2007]:

> (with agile) instead of investing months in building detailed software requirements specifications . . . teams focus on delivering early, value-added stories into an integrated baseline. Early delivery serves to test the requirements and architectural assumptions, and it drives risk out by proving or disproving assumptions about integration of features and components.
>
> No longer do management and the user community wait breathlessly for months, hoping the team is building the right thing. At worst, the next checkpoint is only a week or so away, and . . . users may be able to deploy even the earliest iterations in their own working environment.

So, with agile, we take a far more flexible approach to requirements management: one that is far more temporal, interactive, and just-in-time. Gone are the traditional software requirements specifications, design specifications, and the like, and, along with them, the implied commitment to deliver "all that stuff" on a fixed schedule and fixed resource basis.

Goodbye Iron Triangle

The net effect of this change is to eliminate the iron triangle that has kept us from achieving our quality and dependability objectives. In the agile battle of date versus scope, the date wins. In other words, with agile methods, we'll fix two things, schedule and resources, and we'll float the remainder, scope (requirements), as the DSDM-inspired Figure 1–8 illustrates.[6]

---

6. Dynamic System Development Method: An agile method with roots in RAD

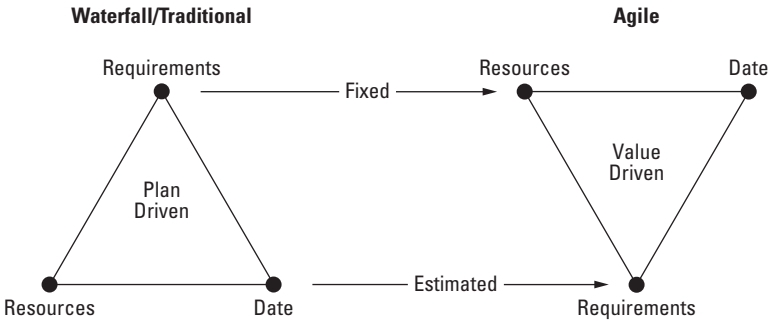


Figure 1–8 Agile fixes the date and resources and varies the scope.

Also, as we apply the appropriate agile technical practices, quality is also fixed. So, now we have a truly virtuous software cycle:

Fix quality—deliver a small increment in a timebox—repeat.

Agile Optimizes ROI Through Incremental Value Delivery

Agile is also based on a simplistically sound economic principle—the sooner we deliver a feature, the sooner our customers will pay us for it. This improves the return on investment for the cost of development, as Figure 1–9 illustrates.

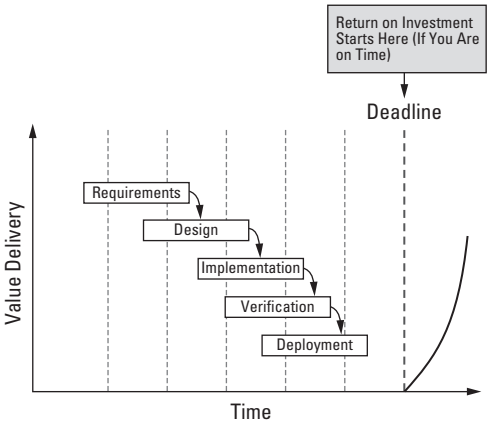


Figure 1–9a. Waterfall Return on Investment

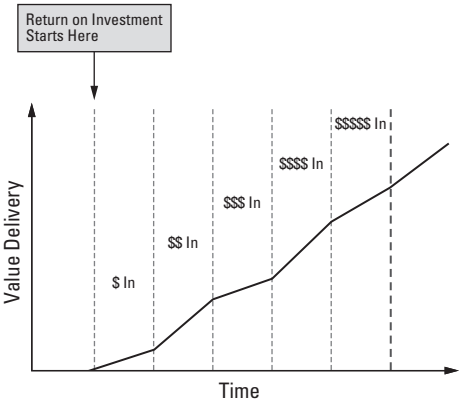


Figure 1–9b. Agile Return on Investment

Figure 1–9 Value delivery and ROI in waterfall versus agile

In waterfall (Figure 1–9a), investment (cost) starts immediately and continues until delivery is reached. *No* return on investment is possible until such time as all committed requirements have been delivered to the customer, or the deadline is reached.

In agile (Figure 1–9b), value delivery starts with the first shippable increment. Therefore, whether business value is measured in customer retention or incremental pricing, return on investment starts then too.

If we assume the investment is constant (though, as we'll see later, the actual investment will actually be much lower in agile), then this is true:

ROI $$ (agile) > ROI $ (waterfall)

Wait, It Gets Even Better

However, even this simple example understates the case for increased ROI because it doesn't take into account the differential value of early market features. For example, when I bought my first iPhone, the initial price was an eye-popping $600. As an early adopter, I purchased mine within just a few months of launch and paid the full price. I did so because it was the only product on the market at that time that offered the feature set of a full-touch UI, integration with iTunes, and the promise of the future applications store. And I wanted it! Twenty-four months later, you could buy a much more powerful version, with 3G data network, integral GPS, video, and a host of other features, for about $199, which is *one-third the price* I paid just two years earlier for about *half* the capability.

Anyone entering the market later with a "me too" product had to compete at a much lower price. Moreover, they had to invest heavily to disrupt an incumbent market of early adopters who are unlikely to switch as the iPhone makes its way into its users' daily lives.

What this story describes is a well-known causal market ROI behavior:

> The value of any marketable feature decreases over time.

Therefore, to capture the maximum gross profit, you have to be in the market first, or at least early enough to where the value/pricing differential is still in effect. When you superimpose that phenomenon on the curves of Figure 1–9, you get a truly startling effect. ROI actually increases at a rate even faster than the linear rate implied by Figure 1–9a, as is illustrated in Figure 1–10.

Taking this into account, the following becomes clear:

ROI $$$ (agile) >> ROI $ (waterfall)

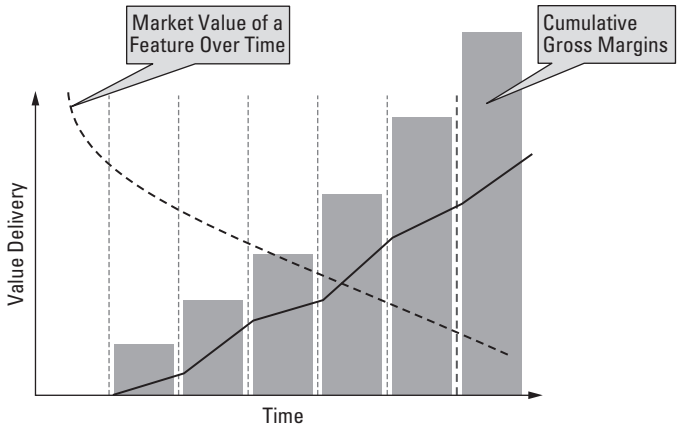
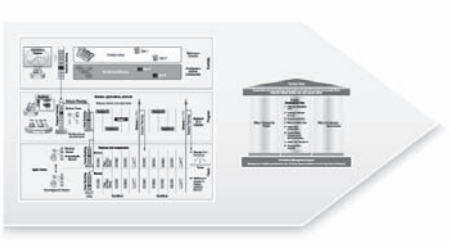


Figure 1–10 Agile ROI, taking into account differential feature value over time

Now we see why an already successful enterprise may be willing to transform itself to more agile practices, even in the face of serious political, operational, governance, and organizational impediments:

Because it makes economic sense to do so.

ENTERPRISE-SCALE ADAPTIVE PROCESSES



Enterprise-Scale Adaptive (Lean & Agile) Processes

The compelling nature of this ROI data, coupled with the increases in productivity, quality, and morale achieved by agile teams, is incenting larger software enterprises to adopt agile methods. Although the methods we've described so far were developed in smaller team contexts, they are now being actively applied and extended in larger enterprises worldwide. Books such as *Scaling Software Agility: Best Practices for Large Enterprises* [Leffingwell 2007] and *Scaling Lean and Agile Development* [Larman and Vodde 2009] are adding momentum to this movement by providing additional practice guidance around the broader enterprise topic areas including organization, product line and systems architectures, governance, and portfolio management.

INTRODUCTION TO LEAN SOFTWARE

In a parallel universe, the roots of the lean software movement were evolving primarily from the successes of Toyota and the Toyota Production System (TPS), an alternative to the mass production systems in use at the time. TPS is a set of lean, economically-based manufacturing philosophies, principles, and practices used to vault Toyota to become the world's leading car manufacturer by 2007. This method has been described in a number of books, including classics such as *The Toyota Way* [Liker 2004].

In turn, this has spawned a growing *lean software* movement, (note: "lean" appears with about 3% market share in Figure 1–7) which is being promulgated by thought leaders in books such as *Implementing Lean Software Development* [Poppendieck and Poppendieck 2007], *Lean Software Strategies* [Middleton and Sutton 2005], and *Lean-Agile Software Development* [Shalloway 2010]. In addition, lean thinking has been applied successfully in product development in books such as *Managing the Design Factory* [Reinertsen 1997] and *Principles of Product Development Flow* [Reinertsen 2009]. The move to lean thinking in software and systems development is now also being actively supported by a body of knowledge and certification body, the Lean Software and Systems Consortium.[7]

With respect to product development, lean thinking is based on an extensive set of proven economic and mathematical principles that describe the flow of product information within the enterprise but that apply equally well to the supplier and customer elements of the larger business value chain. As such, it is broader and deeper than the specific agile software methods we have described so far.

Indeed, it is easy to view XP, Scrum, and others as "software instances of lean," and as such, lean provides an even broader framework for improving the economics of new product development in those enterprises dependent on software.

The impact of lean thinking is only beginning to be felt in the industry today, but it seems likely that the impact of lean over time will be as great or greater than the effect of the agile software development methods we have described so far. Therefore, we'll take some time to establish the framework for lean software thinking in the following sections, because this set of principles also underlies the premise for our approach to *lean requirements practices for teams, programs, and the enterprise*.

The House of Lean Software

As we mentioned, the principles and practices as applied to lean manufacturing, lean product development, lean services, and lean thinking in general are deep and extensive.

7. *www.leanssc.com*

Although the general body of work is enormous, Larman and Vodde[8] have described a framework for lean software thinking that translates many of the core principles and practices into a manageable software context. In so doing, they also reintroduced a "house of lean thinking" graphic, inspired by earlier houses of lean from Toyota and others. Perhaps because I'm a visual learner, I've always liked that graphic, so I've created a variant for our "house of lean software," which is illustrated in Figure 1–11.

Our house of lean software has five elements:

- *Roof, the Goal:* Sustainably delivering value fast
- *Pillar 1:* Respect for people
- *Pillar 2:* Continuous improvement
- *Foundation:* Management support
- *Contents:* Product development flow

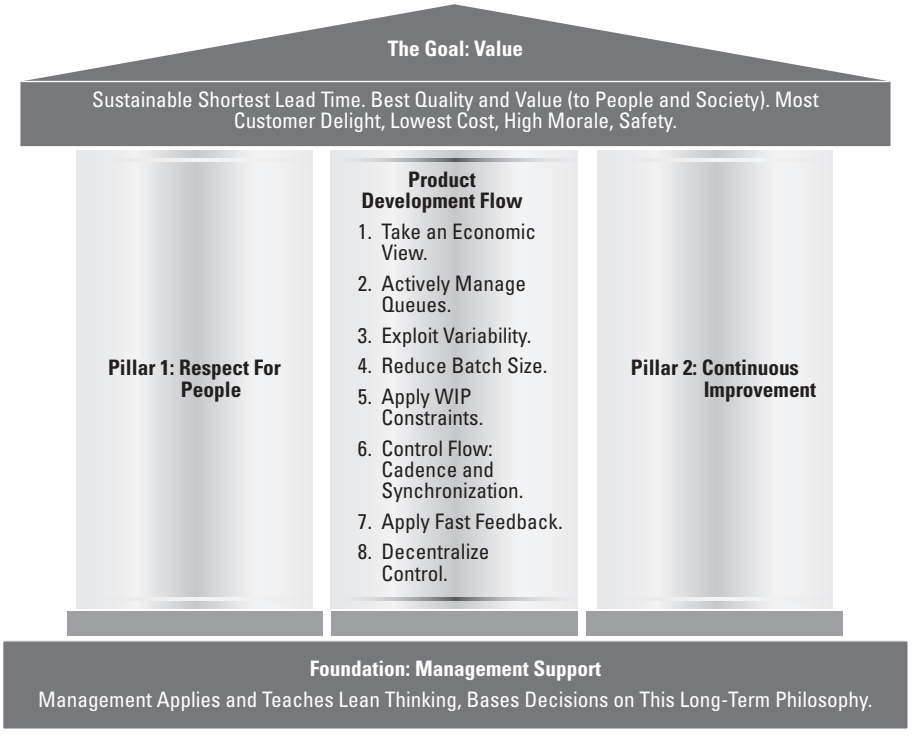


Figure 1–11 House of lean software

Adapted from Toyota Production System [Liker 2004], *www.leanprimer.org* [Larman and Vodde], Reinertsen 2009

8. *www.leanprimer.com/downloads/lean_primer.pdf*

The first four elements—the roof, pillars 1 and 2, and the foundation—provide the philosophical framework for lean software thinking. The fifth element, the product development flow, describes the specific lean principles we'll apply throughout this book.

We'll describe each of the elements of the house of lean in the following sections.

Roof, the Goal: Sustainably Delivering Value Fast

The goal of lean is unarguable: to deliver the maximum amount of value to the customer in the shortest possible time frame. Here's how others put it:

> *All we are doing is looking at the timeline, from the moment the customer gives us an order to the point where we collect the cash. And we are reducing the timeline by reducing the non-value-added wastes.*"
>
> —Taiichi Ohno

> "*We need to figure out a way to deliver software so fast that our customers don't have time to change their minds.*"
>
> —Poppendieck

> "*Focus on the baton, not the runners.*"
>
> —Larman and Vodde[9]

So in our requirements work, we are reminded to do the following.

- Focus on customer requirements as they move through the system, rather than the people and organizations who manage them.
- Search for, and actively minimize, delays, handoffs, and other non-value-added activities.

9. Craig Larman commented: "In the Scrum-origins paper [The New, New Product Development Game, Harvard Business Review, 1986] the authors emphasize the importance of no handoff, and refer to the best approach as *not* handing off a ball or baton to other (specialist) teams, but rather, a cross-functional team that 'moves the ball down the field together' (rugby metaphor). [however] realistically today, there are still going to be handoffs—between sales, product mgmt, R&D, manufacturing, and operational support. Given this large-scale challenge, the 'watch the baton' (focus on the value flow and value/waste ratio, not the busy-ness of people or local optimization) viewpoint is still relevant to 'get' the lean-thinking viewpoint."

In addition, we must remember that requirements are not an end unto themselves. We don't really care if we've done a good job of discovering, organizing, prioritizing, and managing them. We only care how they ultimately serve us as the carriers of value delivery through the enterprise, from "concept to cash" [Poppendieck and Poppendieck 2007]—a proxy if you will—for what the customer needs and wants. In turn, all the associated process mechanisms we use to deliver value in our product, system, or service must serve that ultimate purpose.

From a software requirements perspective, we can visualize our enterprise's software delivery *value chain* as in Figure 1–12.

To optimize delivery time and increase ROI, we'll need to optimize the value chain of requirements-to-code-test-delivery by optimizing each of these functions. And to really accelerate value delivery, we will also need to minimize all the delays implied by the whitespace between.

Pillar 1: Respect for People

Although it is fair to "focus on the baton" (requirements), we must constantly be aware that it is our people who actually *do* all the value-added work, and respect for people is a comforting and fundamental principle of lean and agile.[10]

In addition, people are empowered in lean to evolve their own practices and improvements. Management challenges people to change and may even ask what to improve, but workers learn problem-solving and reflection skills and decide for themselves how to make the appropriate improvements.

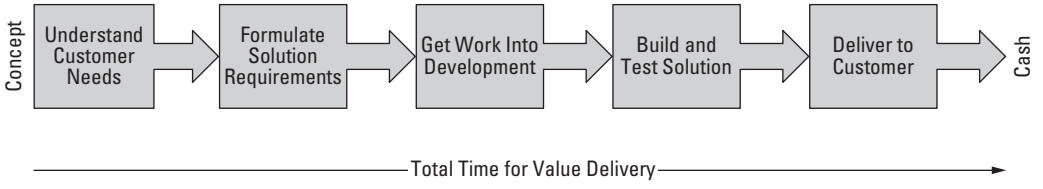


Figure 1–12 A software delivery value chain

10. Agile Manifesto synonym: *Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.*

Pillar 2: Continuous Improvement

This leads to the second pillar of lean, continuous improvement, or *kaizen*. With kaizen, we are guided to "become a learning organization through relentless reflection and continuous improvement." [Liker 2004][11]

- Solve problems and improve processes by going to the source and personally observing and verifying data.
- Make decisions slowly by consensus, thoroughly considering all options; implement decisions rapidly.
- Use continuous improvement to determine the root cause of inefficiencies and apply effective countermeasures.
- Protect the organizational knowledge base by developing stable personnel, slow promotion, and careful succession systems.
- Reflect at key milestones and after you finish a project to openly identify all the shortcomings of the project.

Foundation: Management Support

The foundation of lean thinking is management support. In fact, *support* is an inadequate word to describe the key and active role that management takes in implementing and driving lean. *Leadership* would be a better word. In lean, management is trained in the practices and tools of lean thinking and continuous improvement, applies them routinely, and teaches employees how to use them as well.

In this case, lean deviates from much of our experience with agile development. In our experience, agile has often been promoted as a team-based process that, in the worst case, tends to exclude management from key process and practices.[12] Of course, excluding management from participation and problem solving does not scale very well, and here is a key differentiator between agile and lean that we can leverage.

- In agile, it has been our expectation that management supports us and helps eliminates impediments.
- In lean, the expectation is that management leads us, is competent in the basic practices, and takes an active role in driving continuous improvement.

---

11. Agile Manifesto synonym: *At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.*
12. In the Scrum story, there are "chickens" and "pigs." Those who actively write and test the software are pigs (fully "committed" to the "ham and eggs restaurant partnership"). Others, including management, are "chickens" and are only "involved." This otherwise cute story has led to a tendency in agile to assume that management is somehow not as committed as the team and is therefore not as necessary; perhaps they are not needed at all This is not helpful because it exacerbates organizational silos and inhibits successful adoption.

This key principle is one of the major drivers for lean in the software enterprise. Managers and executives lead, rather than follow, and are accountable for continuously advancing practices.

Contents: Principles of Product Development Flow

In the center of the house of lean software are the various principles and practices teams use to actually develop and deliver software to their end users. Of course, there is no one right way to do this, and the reference books mentioned earlier provide varying perspectives.

In his latest book, *Principles of Product Development Flow*, Reinertsen [2009] calls out eight key themes, each supported by a number of supporting principles. Together, the themes and principles provide comprehensive guidance for lean, flow-based product development. In my view, this is the best and most general description of lean principles as applied to product development, and by extension, they provide excellent, though nontrivial, guidance to the software development team. Reinertsen's book is a rigorous treatment, which contains more than 175 supporting lean and flow principles, so we can't possibly describe them here.

What we can do, however, is introduce the eight high-level themes and then apply various supporting principles throughout this book. The eight themes are as follows.

- Take an economic view.
- Actively manage queues.
- Understand and exploit variability.
- Reduce batch sizes.
- Apply work-in-process (WIP) constraints.
- Control flow under uncertainty—cadence and synchronization.
- Get feedback as fast as possible.
- Decentralize control.

Because these provide much of the underlying lean philosophy in this book, we'll describe each here.

- *Take an economic view:* Take an economic view to establish the decision framework for your specific context, whether it is at the team, program, or enterprise level. Understand the full value chain. Do not consider money already spent. Sequence high-risk, low-cost activities first. If you quantify only one thing, *quantify the cost of delay.*
- *Actively manage queues:* Long queues are universally bad because they create longer cycle times, increase risk, lower quality, and decrease motivation.

Actively manage queue lengths and provide predictable wait times by applying Little's law.[13] Operate at below-peak levels of utilization to increase responsiveness to change. Use cumulative flow diagrams to manage throughput.

- *Understand and exploit variability:* In manufacturing, variability must be minimized to create predictable results, efficiency, and quality. In software development, variability is inherent. Instead of eliminating variability, we must design systems that expect and address variability and even exploit it when appropriate.
- *Reduce batch sizes:* Large batch sizes create unnecessary variability and cause severe delays in delivery and quality. The most important batch is the transport (handoff) batch between teams and between roles within a team. Optimize proximity (co-location) to enable small batch sizes. Good infrastructure (test automation, continuous integration, and so on) and loose architectural coupling enable delivery of software in small increments (batches).
- *Apply WIP constraints:* The easiest way to control queue length is to apply constraints to work in process. Limiting work in process helps force the input rate to match available capacity. Timebox deliveries to help prevent uncontrolled expansion of work. Constrain global WIP pools by constraining local WIP pools. When WIP is too high, purge lower-value projects. Make WIP continuously visible (whiteboards and sticky notes).
- *Control flow under uncertainty—cadence and synchronization:* Even in the presence of variability and uncertainty, we can keep our software process in control with cadence and synchronization. *Cadence* is a predictable rhythm that helps us transform unpredictable events into predictable events. This makes waiting times predictable, lowers transaction costs, and increases dependability and reliability of the product development process. Periodic *resynchronization* allows us to limit variance and misalignment to a single time interval. Regular, system-wide integration (component synchronization) provides high-fidelity system tests and objective assessment of project status. Regular synchronization also facilitates cross-functional trade-offs and high-bandwidth information transfer.
- *Get feedback as fast as possible:* Software development cannot innovate without taking risks, and we need fast feedback to take fast corrective action. Fast feedback (short iterations, short release time frames, fast continuous integration, minimal delays between code and test, and so on) has many

13. Little's law tells us that the average waiting time in a queue is equal to the average length of the queue divided by the average processing rate. Long queues and slow cycle times beget long waits.

benefits: truncates unsuccessful paths quickly, reduces the inherent cost of failure in risk taking, and improves the efficiency of learning by reducing the time between cause and effect. Fast feedback is facilitated by small batch sizes but often requires increased investment in the development environment to understand smaller changes.

- *Decentralize control*: The faster we go, the less practical it is to have decisions move up and then back down the chain of command. Delays in decision making slow feedback and simultaneously decrease the fidelity of the decision, because of the decay in fact patterns that occur in the waiting time. Teams must be empowered to make decisions and act quickly and efficiently. There is little danger because the faster the feedback, the faster even a poor decision can be corrected.

These eight themes provide an economic, quantitative, and mathematically proven substrate for lean and agile software lean requirements management. However, there is one final theme that must be explored before we get on the work at hand.

A Systems View of Software Requirements

> *What we need to do is learn to work in the system, by which I mean that everybody, every team, every platform, every division, every component is there not for individual competitive profit or recognition, but for contribution to the system as a whole on a win-win basis.*
>
> —W. Edwards Deming

Lean thinking requires a *systemic* approach to managing operations throughout and across all the components (departments, artifacts, practices and processes, individuals, and so on) of the enterprise. Optimizing the behavior of any one function, such as requirements management or of a role such as the product owner or product manager or even an entire agile team or business unit—will not produce an optimum, system-level result. Rather, we must look beyond the project team and recognize and optimize all the facets of our requirements process in a comprehensive and systematic way.

For example, we must understand how our new agile requirements model impacts the definition and development of the enterprise-class systems architectures that are necessary to host the new value proposition. In addition, we'll need to understand the impact of requirements practices on departmental and organizational activities—not just at the individual project or team level but all the way to the enterprise portfolio—because that is where the new projects are formed.

We'll cover these perspectives in Part IV of this book.