Chapter 2

Why the Waterfall Model Doesn't Work

Moving an enterprise to agile methods is a serious undertaking because most assumptions about method, organization, best practices, and even company culture must substantially evolve. While we have noted the benefits that agile software methods provide, we have yet to describe how it is that agile can so dramatically produce these results. For perspective, we first look at the traditional software model, the waterfall model of development, and see why this apparently practical and logical approach to software development fails us in so many circumstances.

Figure 2–1 illustrates the traditional model of software development and its basic phases of requirements, design, coding and test, system integration, and operations and maintenance.

This model of development has been with the industry for over 30 years. Many believe that the original source of this model can be found in the paper *Managing the Development of Large Software Systems* by Winston Royce [1970]. However, we also note that Royce's view of this model has been widely misinterpreted: he recommended that the model be applied *after* a significant prototyping phase that was used to first better understand the core technologies to be applied as well as the actual requirements that customers needed! As Winston's son, Walker Royce, now with IBM Rational, points out [Larman and Basili 2003],

> *He was always a proponent of iterative and incremental development. His paper described the waterfall as the simplest description . . . the rest of the paper describes [iterative practices].*

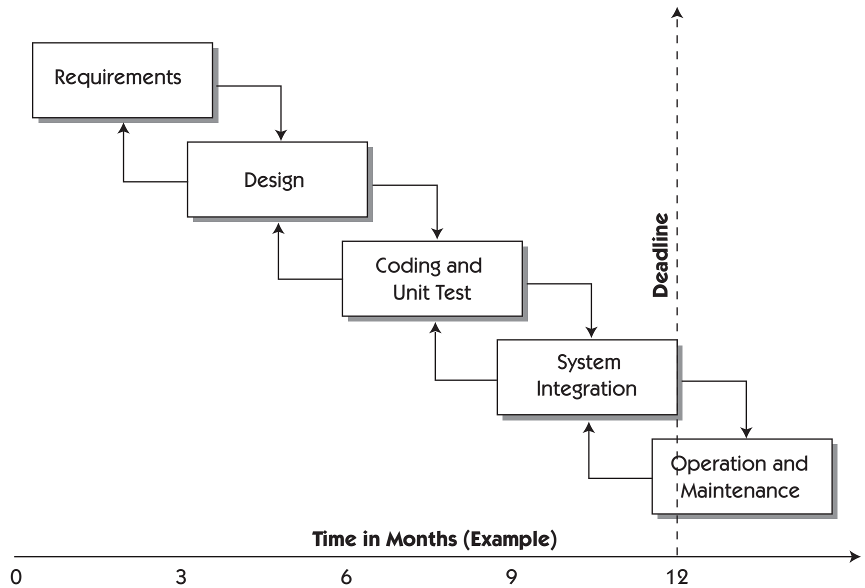

Figure 2–1 The waterfall model of development

Even though misinterpreted, this model of development was a substantial improvement over the former model of "code-and-fix, code-some-more, fix-some-more" practices that were common at the birth of our software industry. While it is in vogue today to belittle the waterfall model for its ineffectiveness and the rigidity that the model implies, it is also true that tens of thousands of successful applications were built using this basic model: first, understand the requirements; second, design a system that can address those requirements; and third, integrate, test, and deliver the system to its users.

Although we accomplished great feats, many projects also suffered greatly because our ability to predict when we would deliver quality software became suspect at best. Late delivery was common; delivering solutions that both were late and did not fundamentally address the customer's real needs was also common. Over time, we all came to understand that the "dreaded system integration phase" would not go as well as planned.

On the upside, however, the model provided a logical way to approach building complex systems, and it also provided time for the up-front requirements discovery, analysis, and design process that we knew we needed. For these

reasons, this traditional model is still in wide use today, and as we will see throughout this book, the concepts of the model never really leave us, but the application of those concepts changes dramatically indeed.

PROBLEMS WITH THE MODEL

Most practitioners reading this book already understand that the waterfall model does not produce the results we desire—there is unlikely to be any other motivation to read the book! Since at least the 1980s, practitioners such as Barry Boehm, with his spiral model of development [Boehm 1988], have strived to amend the model in ways so as to produce better outcomes. In his book *Agile and Iterative Development: A Manager's Guide*, Larman [2004] describes many of the basic principles of software agility as espoused by the various agile and iterative methods. He also takes the time to highlight some of the statistical failures of the waterfall model. For solid, statistical evidence of how this model fails us again and again, we refer you to Larman's book, though we summarize a few key statistics here to illustrate the need to change.

For example, in one key study of 1,027 IT projects in the United Kingdom, Thomas [2001] reported that "scope management related to attempting waterfall practices was the single largest contributing factor for failure." The study's conclusion:

> *[T]his 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, and that spending significant time and effort in fighting the to maintain the maximum level is inappropriate.*

Larman also notes that other significant evidence of failure in applying the waterfall comes from one of its most frequent users, the U.S. Department of Defense (DoD). Throughout the 1980s and into the 1990s, most DoD projects were mandated to follow a waterfall cycle of development as documented in the published standard DoD STD 2167. A report on failure rates in one sample concluded that 75 percent of the projects failed or were never used. Consequently, a task force was convened. Chaired by Dr. Frederick Brooks, a

well-known software engineering expert, the report recommended replacing the waterfall with iterative and incremental development:

> *DOD STD 2167 likewise needs a radical overhaul to reflect modern best practice. Evolutionary development is best technically, it saves time and money.*

In addition, Larman [2004] notes that Barry Boehm published a well-known paper in 1996 summarizing failures of the waterfall model and suggested the use of a risk-reducing, iterative and incremental approach. He combined that approach with three milestone anchor points around which to plan and control the project; this suggestion was eventually adopted as the basis of the Unified Process.

Statistical evidence and our own experience lead us to conclude that the waterfall model does not work. And yet to understand the roots of agility, we must look a little harder at the waterfall model *to understand why it failed us and how agile methods directly address the causes of these failures.*

Assumptions Underlying the Model

In retrospect, it appears that we made at least four key assumptions with the model that simply turned out to be incorrect:

1. There exists a reasonably well-defined set of requirements if we only take the time to understand them.
2. During the development process, changes to requirements will be small enough that we can manage them without substantially rethinking or revising our plans.
3. System integration is an appropriate and necessary process, and we can reasonably predict how it will go based upon architecture and planning.
4. Software innovation and the research and development that is required to create a significant new software application can be done on a predictable schedule.

Let's look at these assumptions in light of what we have learned over the last few decades of building enterprise class systems.

Assumption 1: There Exists a Reasonably Well-Defined Set of Requirements if Only We Take the Time to Understand Them

We now realize that there are a number of reasons why this assumption is false:

- *The nature of tangible intellectual property.* The software systems that we craft are unlike the mechanical and physical devices of the past. We have a history of developing tangible devices, and their physical instantiations allow people to relate to their form and function. Software, however, is intangible. We envision solutions that we think will work; customers concur that our vision appears to make sense—while they simultaneously envision *something somewhat different*. We write down those elements in relatively formal constructs, such as software requirements specifications, that are difficult to write and read, and then use that flawed understanding to drive system development.
- *The "Yes, But" syndrome.* Leffingwell [Leffingwell and Widrig 2003] describes the "Yes, But" syndrome as the reaction that most of us receive when we deliver software to a customer for the first time: "*Yes*, I see it now, *but* no, that's not exactly what I need." Moreover, the longer it takes for us to deliver software to the end user, the longer it takes to elicit that reaction and the longer it takes to evolve a solution that actually addresses the user's needs.
- *Changing business behavior.* We also discovered an even more pernicious aspect of providing software solutions: delivery of a new system changes the basic requirements of the business and the behavior of the system, so the *actual act of delivering a system causes the requirements on the system to change.* The longer we wait to discover the change, the greater the change is going to be and the more rework will be required.

Assumption 2: Change Will Be Small and Manageable

Figure 2–2 illustrates how requirements tend to evolve and change over time. While the data on this curve is neither calibrated nor absolute, the figure reinforces our experience that requirements *do* change *while* the system is being developed.

Moreover, the longer the time between determining the requirements and delivering the system, the more substantial the change will be. If development were really fast and changes were really small, we could track our market very

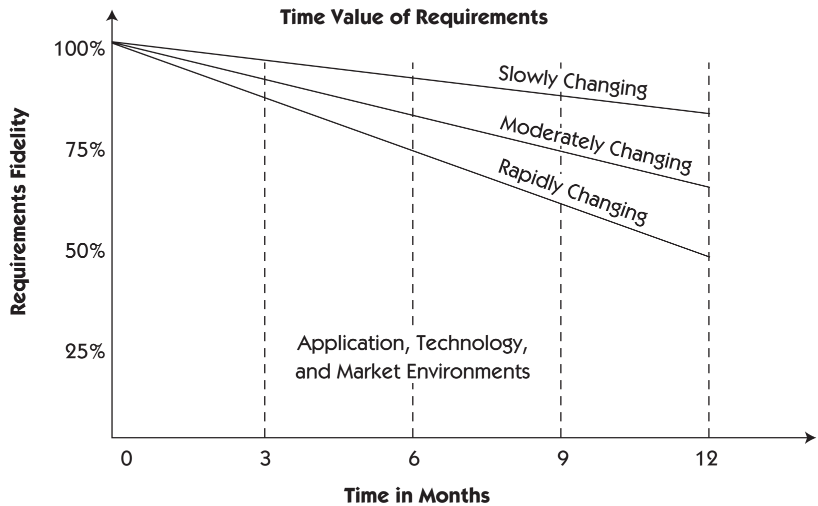

Figure 2–2 Time value of requirements

closely. But if development is slow and change happens fast, bad things are bound to happen.

Assumption 3: System Integration Will Go Well

This assumption underlies our premise that, with appropriate planning and analyses, we could predict how all the components of a complex system would come together and interact. We assumed that our architectural plans would address any system integration problems; that volumes of code written by largely independent teams would work well together; and that overall system behavior, including performance and reliability, could somehow be predicted by our plans and models. To address this challenge, we sometimes even created entire teams or departments—system integration teams—responsible for bringing all of those components together and testing at the system level.

Well, of course, system integration didn't go well, and we discovered that many of our assumptions about the way the system needed to work were incorrect. Our faulty assumptions often led to significant rework on a wholesale basis, further delaying the system and setting us up for another difficult system integration attempt somewhere down the road. The lesson learned is that even a relatively thorough up-front analysis could not predict nor control the system integration process. The problem is too complex; change hap-

pens midstream; technologies evolve during the project and integration assumptions are often wrong and are simply discovered too late.

Assumption 4: We Can Deliver on Schedule

Given the time for proper planning, we assumed that we could know enough about the system, its requirements, and the resources required to build it, and that we could reliably predict *when* we were going to deliver a system. Underlying this assumption are many more:

- We have planned for *what we know*, and we have left sufficient allowance in our plans for *what we don't know*.
- We have allowed room in the schedule to recover from unpredictable events.
- We are good at estimating software development in general, and therefore our prediction in this case is reasonably accurate.

The most telling assumption of all is that we can do it *once*, and we can do it right *the first time*.

In other words, we assumed that we could schedule innovation, and even software research, in a predictable way. We have now proven that this is not the case.

In my own personal analyses and studies of software projects over the last decade or so, I've concluded that teams actually *can* estimate—just not very well—and that their estimates are typically off by approximately *a factor of at least two*. In other words, even with a known, fixed set of resources and even when facing a new application in a known domain, *teams will typically take twice as long as they estimate to reach the desired result*.

This is partly attributable to the complexity of the planning in waterfall projects being exponentially harder than it seems because different parts of the system (or parts of the team) transition between phases at different times. So, essentially we have "dozens, and perhaps even hundreds," of parallel waterfall models proceeding (not exactly in parallel) toward a final result. How does a team go about integrating *that*?

In the case of a twofold schedule slippage, what happens to our model and our deadline (not to mention our team!) is illustrated in Figure 2–3.

Well, perhaps it is unfair to assume that these phases are truly linear in time, but experience has shown that, in all likelihood, we will be at or near the

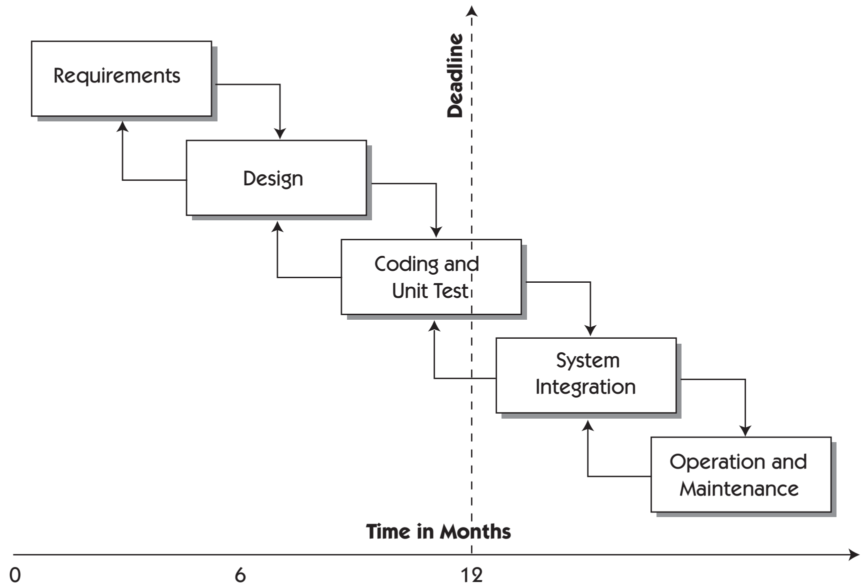

Figure 2–3 What happens when everything except the deadline moves to the right

system integration phase when the deadline passes. Large portions of the coding may be complete, but perhaps not a lot of it has been tested, and almost none of it has been tested at the system level. In addition, it is likely that dozens, and even hundreds, of individual feature or requirement threads are proceeding in parallel.

Even if these individual threads have been tested at the component level (which may be unlikely because component-level testing was also likely deferred until closer to this point in time), no system-level testing has been done, as doing so requires a level of completeness necessary to support integration. Moreover, as we perform the integration during system-level testing, we are discovering that the system does not work as anticipated.

The inevitable scope triage (and typically promotion of nonparticipants) happens at that point, but it is far too late. In our attempt for recovery, we are severely hampered by the following facts:

1. Much investment has been made in internal infrastructure and features that do not work holistically at this point. Backing this code out is not an easy task: we cannot simply leave out troublesome require-

ments, because interdependencies have already been built into the system. Scoping out requirements at this late date causes us to rework some work in progress, which in turn causes *further investment in elements that are not going to be delivered in this release*—a wasteful process to the extreme.

2. Even worse, because we're not yet complete with system integration, *we haven't even discovered all the issues inherent with the assumptions that brought us to this point*, so even with severe scope management, much of the risk still lies ahead of us.

Surely we are in a difficult spot now: the deadline is here; we are going to have to do a lot of work to eliminate things that we've already invested in; we haven't as yet actually performed enough integration to know that the system in the end will work. And, in the worst case, we may have nothing useful to ship to our customer for evaluation, so we don't even yet know if the system that we've intended to build will meet the real requirements!

Wait—It Gets Worse As if the situation weren't bad enough already, let's look at the picture of requirements decay that we've experienced during this time frame (Figure 2–4).

So, if it really took us twice as long as we thought to get to this point, no matter the rate of decay, the requirements will have changed twice as much as we

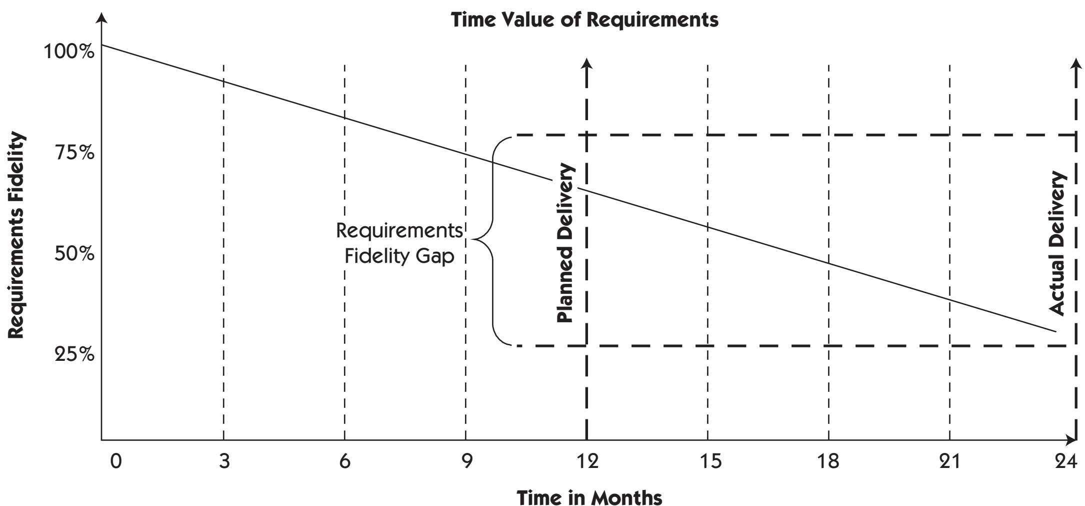

Figure 2–4 Planned versus actual requirements fidelity gap

thought they might by the time we reach this point. (If we thought we would see 25 percent requirements change, we really see 50 percent change.) This means that the system we are now building is not nearly as good a match to the target requirements as we anticipated.

In Conclusion

- We failed to deliver the application as intended to the customer in the time frame we predicted.
- We now understand that the solution that we have in process is somewhat off the mark.
- We likely don't have anything to ship to the hold the customer's confidence.
- Reworking what we have may require ripping out some of what we've already built.
- We haven't driven the risk out of the project because integration is not complete.

If large-scale systems development using this simple, compelling, and yet fundamentally wrong model can result in such grand misfortune, then there has to be a better way.

Enter Corrective Actions via Agile Methods

How do agile methods fundamentally address the flawed assumptions of our former model?

Agile avoids virtually all of these underlying assumptions of the waterfall model and addresses these problems in the following ways:

1. We do *not* assume that we, or our customers, can fully understand all the requirements, or that anyone can possibly understand them all up front.
2. We do *not* assume that change will be small and manageable. Rather, we assume that change will be constant, and we deliver in small increments to better track change.
3. We *do* assume that system integration is an important process and is integral to reducing risk. Therefore, we integrate from the beginning and we integrate continuously. We live under the mantra "the system always runs," and we strive to assure that there is always product available for demonstration and/or potential shipment.

4. We do *not* assume that we can develop new, state-of-the-art, unproven, innovative, and risk-laden software projects on a fixed-functionality and fixed-schedule basis. Indeed, *we assume that we cannot.* Instead, we assume that we can deliver the most important features to our customer earlier, rather than later, than the customer might have expected. In so doing, we can get immediate feedback on whether we are building the right solution. If through immediate and direct customer feedback we discover that it is not the right solution, all is not lost. A much smaller investment has been made to this point, and we can refactor the solution and evolve it rapidly, while delivering continuously, and we can do so without excess rework.

In conclusion, agile makes an entirely new set of assumptions about the fundamental nature of this innovative process we call software development. In so doing, we change the basic paradigm: we move to rapid delivery of the highest priority requirements, followed by rapid feedback and rapid evolution, and we are thereby better able to deliver a solution that truly meets the customer's end goals.

If such is the power of agile, it's time to take a harder look at these methods and understand how we can apply them at enterprise scale.

This page intentionally left blank