1

The Crisis in Software: The Wrong Process Produces the Wrong Results

YOUR ORGANIZATION—WHETHER business, governmental, or nonprofit—likely needs to be able to create value by building, customizing, and using software. Without software, your ability to achieve your goals as a business leader is inherently limited, if not impossible. But despite this need, software development has historically been an unreliable, costly, error-prone endeavor.[1] This leaves you in a pickle: You need software, but you can't get what you need, when you need it, at a cost that is acceptable, at a level of quality the makes it usable.

Indeed, the Standish Group's 2011 CHAOS Report found that more than half of software projects conducted between 2002 and 2010 were either described as challenged or complete failures; just 37 percent were classified as successful (Figure 1.1) The Standish Group modestly defined a successful project as delivering all the requested functionality, on the expected date, for the

[1] April 11, 2005, Forrester Report "Corporate Software Development Fails to Satisfy on Speed or Quality." Corporate development shops continue to disappoint: A fall 2004 Forrester survey of 692 technology influencers—those who hold the information technology (IT) purse strings—indicated that nearly one-third are dissatisfied with the time it takes their development shops to deliver custom applications, and the same proportion is disappointed by the quality of the apps that are ultimately delivered. One-fifth of respondents are unhappy on both counts.

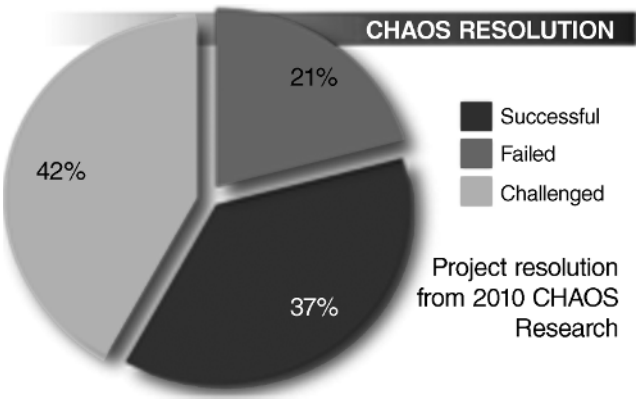

Figure 1.1 Traditional Software Development Is Risky

planned cost. The ability to accommodate changes, the ability to manage risks, or the inherent value of the software weren't considered.

The chances that a software project will be successful are not good. If you are trying to accomplish something critical that involves developing of software, you are probably worried. The software industry has failed you by being slow, expensive, and unpredictable. If software weren't so important, you would probably stop investing in software altogether.

You are not alone. Many others are in the same boat. For example, the Federal Bureau of Investigation's (FBI) Sentinel project recently ran into trouble. The FBI turned Sentinel around using the insights and processes described in this book.

The information here concerning Sentinel comes from the Department of Justice Inspector General reports, and it is publicly available. Before you dismiss this as a corner case, a particularity of government work, think about this: If a large government agency can radically improve how it builds software, then so can your organization.

Case Study: The FBI's Sentinel Project

Every FBI investigation has a case file that contains all of the records that were either created or obtained as part of an investigation. In 2003, the FBI decided to digitize cases and automate the related processes . . . Agents would then

rapidly compare cases and discover connections between them. The name of the project was Sentinel.

In March 2006, the FBI initiated development of Sentinel, targeting an end-user base of more than 30,000 FBI agents, analysts, and administrative employees. Original estimates for Sentinel were $451 million to develop and deploy by December 2009. According to the FBI's original plan, Sentinel was to be developed in four phases. The FBI contracted the work to Lockheed Martin. Lockheed Martin proposed using a traditional software development process.

But by August 2010, the FBI had spent $405 million of the $451 million Sentinel budget but delivered the functionality for only two of Sentinel's four phases. Although these deliverables did improve the FBI's case management system, they did not deliver much of the value that was originally envisioned. Because of the cost and timeline overages, the FBI issued a stop-work order in July 2010 that directed Lockheed Martin to halt all work on the two remaining phases of Sentinel.

To this point, the FBI had been using a traditional development process, and it now chose to adopt a new approach to see if it could obtain better results. We developed this new process, called Scrum, in the early 1990s. The same Standish Group CHAOS Report that classified just 37 percent of projects as successful demonstrated how different the results of a traditional approach are versus those of using an agile, or Scrum, approach (Figure 1.2);

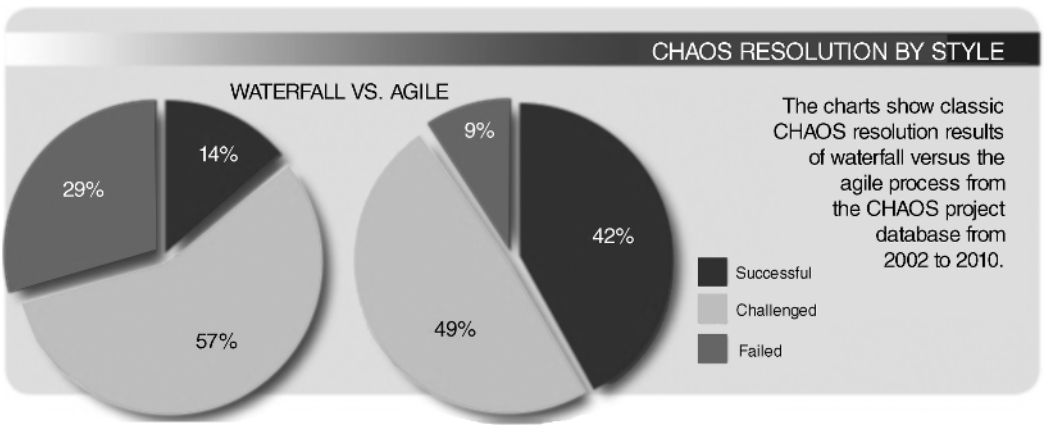

Figure 1.2 Agile Projects Are 3 times as Successful

specifically, it noted that whereas only 14 percent of traditional projects succeeded, a full 42 percent of projects using an agile approach achieved success. We argue that in addition to the Standish Group's traditional definitions of success, these projects also enabled greater responsiveness to changing customer needs, allowed for better risk mitigation, and ultimately delivered better-quality software.

By 2009, the FBI recruited a new chief information officer (CIO) and chief technology officer (CTO) with experience managing organizations that built software using our approach. They decided to see if this more agile approach could help the FBI. In 2010, the CTO told the Department of Justice that he was going to change the approach for Sentinel. He asserted that this new approach would streamline decision-making processes and allow the FBI to deliver Sentinel within budget. The FBI told the Inspector General at the Department of Justice that it believed it would be able to complete Sentinel with the remaining Sentinel budget and within 12 months of recommencing. An audit conducted by Mitre had previously concluded that the FBI would need an additional $35 million and six more years if it had continued with its traditional approach.

The FBI moved the entire Sentinel project to the basement of the FBI building in Washington, DC, and reduced the Sentinel staff from 400 to 45 people, 15 of whom were programmers. The CTO ran the project himself, managing toward a goal of delivering some of Sentinel's functionality every 30 days. Each increment of functionality had to meet all of the final functional and nonfunctional requirements—this was no "first draft" software. Every three months, the FBI would deploy the features that had been built in the preceding three iterations into a field pilot.

By November 2011, within a year of restarting with the new approach, all phases of Sentinel had been completed. The software was deployed to a pilot group of FBI locations, and remaining offices are scheduled to see deployment by June, 2012. The FBI was able to complete Sentinel for $30 million dollars within 12-months, a cost savings of more than 90 percent.

The people at the FBI worked hard on the first few phases of Sentinel, but their approach to software held them back. After the FBI shifted its approach to the one we lay out in this book, they worked just as hard as before, but they were rewarded with vastly superior results. If an organization like the FBI can do this, why can't yours?

The Wrong Approach: Predictive Processes

The process that the FBI originally used for Sentinel was what we refer to as a predictive, or sequential design, process. In fact, until 2005, the majority of software projects used predictive processes. Don't get us wrong; there are certainly circumstances in which predictive processes are more appropriate and could be successful. These circumstances, however, were the exception rather than the rule. If one can establish a complete vision, define all of the requirements of the vision, and then devise the detailed plan to turn the requirements into the vision, then a predictive process will work. But any deviation from the original vision, requirements, or plan creates great project risk. And with business needs and technology changing as rapidly as they do, it is rare that these elements can actually remain static. As a result, and as the Standish Group has reported, 86 percent of the software projects that use predictive processes are not successful. In fact, we consider the use of predictive processes to be the most common cause of problems in software projects.

The organizations that we work with have typically been struggling to increase the success rates for their software projects. They seek our help because they fear that their software organizations are spinning out of control. Their existing process has failed them, and they do not know of an alternative. Their problems with software development create a tremendous amount of waste for their organization, yet they persist because they are dependent on software to remain competitive.

Here is how the executives and managers typically describe the problems they are facing:

1. *Releases take longer and longer.* "Each release is taking more time, effort, and cost to get delivered to its customer(s). Several years ago, a release might have taken 18 months. That same release now takes us 24 months to develop, package, and deploy. Even then, a release is stressful and requires significant effort. We keep spending more but are getting less and less."
2. *Release schedules slip.* "Commitments are made to customers and prospects. Those customers or potential customers are preparing major business initiatives that depend on our release schedule. They need our release, with the functionality we promised, at the exact point in time we promised to deliver it. We usually let them down at the last moment. Their plans are thrown

into disarray, and they lose money and credibility with their customers. We may not get more business from them; they will obviously not act as good references for us; and they may start looking for other vendors."

3. *Stabilization at end of the release takes longer and longer*. "We got really firm with the development organization. We set firm, inflexible dates that they had to be done by. They met these dates by the end of what they call 'code complete' or a 'code freeze.' But the software was unusable. It didn't do what was needed, it didn't perform as required, and it did all of that badly. We couldn't even ship it as a 'beta release' so we could get feedback from a small sample of customers. The defects were so profound that our beta customers refused to participate. We needed another nine months to ship the release, and even then it was shaky and required a lot of hand-holding and apologies."
4. *Planning takes too long and doesn't get it right*. "We figured the releases were taking too long and then slipping because we didn't plan well enough at the start of the work. We didn't get our requirements firmed up and fully developed, and our estimates included more guesses than they should have. To rectify this, we now spend more time planning. New ideas keep coming up. As people review the plans, they find parts that need to be reworked or clarified. We are now spending much more time planning than we used to, but our schedule slips and stabilization periods are still extensive and awful. Despite our significant efforts, changes still come up during development that weren't and couldn't be foreseen during planning."
5. *Changes are hard to introduce mid-release*. "The current process cannot accommodate change. We spent a lot of time planning everything at the beginning, and all the needed work is predicated on the plan. But often something critical has to be included, or a new feature has to be added to close a sale. To incorporate this change, we have to adjust all the work that we have already done to accommodate it. This is very difficult because it is hard to understand the ripple effects of changes in software. Even when it is important, it feels like the amount of time that it takes to fit it in is often a hundred times greater than if we had known of it when we started. But what can we do? If the change doesn't make it into this project or release, it may have to wait up to two more years to be included in the next release."
6. *Quality is deteriorating*. "We know that we shouldn't pressure the developers to get what was planned and changed out on time, but our business is

hurting from the planning, slippage, and change problems. We tell the developers to toughen up and get it out on schedule with everything we planned. Every time we do that the developers accommodate us by cutting the quality of the software or the testing of its suitability. The result is so bad that we either go back into the stabilization phase or we ship an organizational embarrassment."

7. *Death marches are hurting morale*. "We are treating people in a way we would like not to. However, we have commitments and a business to run, so everyone on the project works weekends and long days. Their families and their health suffer. As a consequence, we have trouble recruiting top developers and we lose our best developers to other organizations. Our existing staff is so demoralized that its productivity is slipping despite the increase in hours."

These examples are enough to discourage any executive or manager. Despite 20 years of herculean efforts and massive expenditures in software, by the early 1990s little progress had been made in ensuring the successful outcome of software projects. The process we will describe to you in this book tackles these problems head on.

The Wrong Results: Project Failure

Your use of the traditional, or predictive, software development process is the root cause underlying so many software project failures. The predictive process, also called waterfall, depends on the accuracy of the project plan and its unswerving execution. It depends on:

1. *Requirements* not changing and being completely understood. Any changes in requirements would change the plan, requiring alterations to the plan that create massive ripple effects and frequently rendering already completed work useless. Unfortunately, more than 35 percent of all requirements change during a typical software project. Business customers struggle to fully define these requirements, but the ever-changing marketplace, their incomplete understanding of what they need, and the difficulty of fully describing the anticipated system until it is done make requirement changes inevitable.
2. *Technology* working without any problems. All of the technology the software uses has to perform reliably and as initially planned. Unfortunately,

the project frequently incorporates technologies planners haven't used in the past—either singularly or in combination, or for the same purposes. What's more, technology standards sometimes change during the course of the project.

3. *People* being as predictable and reliable as machines. The plan calls for a specific network of tasks to be completed, each task requiring a defined number of hours from a specifically skilled resource that is given specific well-defined inputs. Unfortunately, the network of tasks starts wobbling whenever requirements change. Even more problematic, people are not machines! People have their good and bad days, different skill levels, and different attitudes and intelligence. Tasks ended up being executed in a very different way than predicted.

The software development industry understands these difficulties and for years has tried to address them by stepping up its planning efforts. Project planning could take as long as the actual development of the software. Massive amounts of work went into gathering requirements, defining architecture, and detailing work plans.

But all of that work was useful only if the plan was based on accurate information that did not change over time. This method is effective when work is well understood and relatively stable and the plan can consequently remain unchanged. When this is not the case, however, the predictive process fails. It is not constructed to cope with the unknown and the unexpected; it is constructed to optimize problems of constraint.

Many traditional manufacturers successfully employ the predictive process model. The payoff for all this upfront work is the repeated execution of the plan, creating car after car or toaster after toaster. There is no similar payoff in software, as a software development plan is executed only once. The very thing that made predictive processes suitable for manufacturing, where a single process cycle will create high volumes of products, make it ill suited for software, where a single process cycle will create just one product.

The Stacey Graph is a useful tool for assessing the certainty or predictability of work.[2] The Stacey Graph measures the certainty versus the unpredictability of various dimensions of work and categorizes where the work

[2] R. Stacey, *Complexity and Emergence in Organizations* (London: Routledge, 2001).

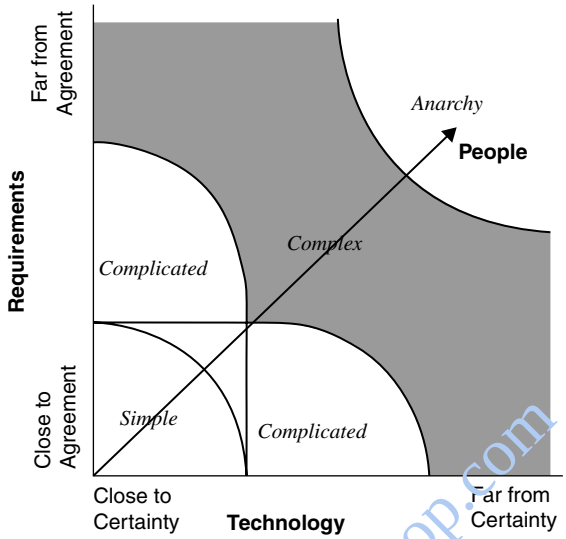

Figure 1.3 The Stacey Graph

falls. We used it to model the three dimensions of software development: requirements, technology, and people, as described in Figure 1.3.

We can plot software development projects as follows:

- *Requirements:* Close to certain with no risk of change, to far from certain, with vague, emergent descriptions and many expected changes.
- *Technology:* Well known and understood, to far from certain, with development and operational technology usually consisting of multiple products interacting through interfaces at different levels of software development and release.
- *People:* Known and constant, with a small number of people on a single team, to software projects involving more than four or five people, often hundreds, who are constantly changing. People by themselves have opinions, attitudes, and moods. Working in groups or teams, the interactions and unpredictability of their work is significant.

Using the Stacey Graph, we can see that software development projects are at least complex and sometimes chaotic. The predictive process, on which waterfall and traditional software development is based, is

applicable only for simple, repeatable work. You can determine whether the right process is being used for your work by the yield rate, the degree of success. If a predictive process were appropriate for software projects, the yield rate (or successful project completion rate) would be very high—about 99.99 percent. However, the Standish report discussed earlier measures a software development yield rate of 14 percent using predictive processes. Most businesses could not survive such a low yield rate. Imagine if General Motors scrapped every seventh car that it built—that's the effect of a 14 percent yield rate.

The predictive process is inappropriate for problems involving software development. Software development is complex, not simple. We assert that the decision to base software development projects on the predictive process was what led us to failure. Our proof lies in the increased yield rate of software development projects when Scrum is employed.

People sometimes equate construction or bridge building with software development. Engineering disciplines, such as bridge construction, fall somewhere between simple and complex on the Stacey Graph. Standardization renders this work only complicated. There are three forms of standardization. Firstly, there are Newtonian laws explaining how physical objects interact. Secondly, standardized materials such as wood beams, metal struts, and fasteners, with standard sizes and known characteristics are employed.. Thirdly, there are standards for all types of construction that are written into codes and inspected for by authorities. None of these things exists in software; what's more, as long as the software industry continues to evolve as rapidly as it has, this is unlikely to change.

Case Study: Parametric Technology Corporation

Parametric Technology Corporation (PTC) is a global firm with 5,000 employees that develops product life cycle management products. These products, which grew out of CAD/CAM (computer-aided design/computer-aided manufacturing) systems, help some of the world's largest engineering organizations—Raytheon, BAE Systems, and Airbus, to name a few—to manage the development of massive systems such as the Airbus A380. They do so this in part by tracking the configuration of all the parts, assemblies, and subassemblies.

In 2005, PTC was suffering from all of the symptoms of predictive software development processes:

1. *Releases took longer and longer*. Releases had crept from 18 months to 24 months, and it looked like the current release was going to take longer.
2. *Release schedules were slipping*. Slips from the initial schedule were up to nine months, and these occurred bit by bit. Customers that relied on the timely delivery of critical functionality were unhappy.
3. *Stabilization at end of the release took longer and longer*. Stabilization was behind at least two-thirds of the slips.
4. *Planning took too long and didn't get it right*. Up to six months was spent planning each release, and even then the plan was wrong and would often have to be changed.
5. *Changes were hard to introduce mid-release*. It was hard to tell if the slips, stabilization, and quality problems were due to changes once the project started, but there were certainly a lot of necessary changes.
6. *Quality was deteriorating*. This was both a serious and increasing problem.
7. *Death marches were hurting morale*. PTC was having trouble recruiting quality people.

PTC's development organization employed a waterfall process, and to make it work better, they had tried to button down on requirements. Requirements were compiled into an exhaustive functional specification document. Only when the requirements were finalized were they shared with the development organization. In the meantime, the developers didn't have much to do. They either fixed bugs or sometimes left PTC out of pure boredom. The quality staff wasn't allowed to start any testing until the product was fully complete, so it had less time to do its work. Working under release date pressure, the quality team was forced to release products with insufficient testing.

Jane Wachutka was the new vice president for product development of PTC's Windchill product. As a new employee, she tried the PTC way of waterfall, and she found that all of the usual problems occurred. At her previous job, she had employed many nonwaterfall techniques similar to those that helped the FBI succeed with the Sentinel project. With this approach, a project consists of one or more iterations of work, each no more than 30 days long. Many small teams of developers selected high-value requirements each

iteration and turned them into an increment of usable software. All the increments of the teams were integrated into one complete and usable increment. In each subsequent iteration, another increment of software was developed and added to the prior increments.

Brian Shepherd, PTC's executive vice president for product development, was skeptical when Jane advocated this new process for building the software in 2007. If he allowed her to do so, she committed to being able to get the programmers started sooner, improving programmer retention rates, engaging the quality group sooner, and not releasing products until they were fully tested and of sufficient quality. Jane stressed that the functional specifications could be imperfect because the product management group would get to see and use parts of the product frequently during the development cycle and give feedback. Brian agreed to proceed with a new process—an agile process called Scrum. He warned Jane, though, "Don't screw it up."

When Jane first told her employees about the new way they were going to develop software, they were skeptical. The individual members of the development teams, in particular, were slow to buy into it. They still struggled to be perfect at each step of development, making sure they were doing exactly what others wanted. However, as they gained some experience in using the new process, the product managers no longer struggled to complete perfect functional specifications before handoff to development; they let the functional specifications emerge throughout the release. Because PTC now developed complete functionality within 30 days, its developers were able to directly collaborate with customers within any appropriate iteration of development. The developers gained insights into the requirements and how they could be best implemented. Customers noticed the differences and began working with development teams during each iteration. The proud customers helped author the functionality and got exactly what they wanted.

The product management team had a rolling three-, two-, and one-year set of requirements. Three years out was the vision, with a description of high-level capabilities. A more detailed picture of which releases would deliver the vision was available for the two-year time frame. For the current year, 30-day iterations were defined for most of the first six months, and there was a road map of goals for the next six months. Each year's set of requirements had more detail than its predecessor. The developers worked on the one-year set of requirements. They called and worked with PTC customers to work out the

details. The entire organization had become a think tank of creativity and productivity.

Within two years, all of Jane's commitments to Brian had been met. Jane's organization was releasing software every 12 months, down from previous release times that exceeded 24 months. The product was of high quality. By 2011, PTC had changed. It had become a transparent organization, both within and to its customers. Surprises rarely happened; customers knew what to expect and when. Defects were low and trending to zero by 2012. New features, user interfaces, and workflow capabilities had been added. The product had been overhauled to make it secure from external threats. Finally, the budget and staffing were both down by more than 10 percent. Brian Shepherd had a new facility built for the software product organization. The space reflected the transparency critical to the new process: Everyone was in an open space, with no offices. All walls were glass.

Recently, Jim Heppelmann, the chief executive officer (CEO) of PTC, listened as his managers positioned themselves for increases in their annual budgets. Finally, he stopped the discussion and asked everyone to thank Jane's organization for reducing costs while improving quality and increasing functionality. Because of them, he said, he could share the savings with others parts of the organization.

In one instance, Jane and Jim were on a conference call to a company in Israel that was evaluating PTC's products. Jane told the CEO that Raytheon was using PTC's products worldwide and she urged him to contact Raytheon's executives. She knew that they were not only impressed with PTC's products; they were thrilled that PTC's new process removed surprises. They were able to collaborate and adjust their schedules in real time with PTC. They were so impressed that they were adopting PTC's way of developing software. Jim jumped in. He told the prospect that Jane had forgotten to mention the last release. It was the best-looking product PTC had ever shipped, primarily because Jane had changed the process.

Summary

Software development has in the past been prone to failure. The root cause of this failure is the use of predictive processes for complex work. When we shift

to Scrum, an empirical process, the software project success rate increases dramatically.

It *is* possible to get software features ready to use in 30 days—or less. Don't let your developers tell you otherwise, because hundreds of thousands of software developers have been doing this since the early 2000s. A software product may still be big, but it can still be built in small pieces, one by one, 30 days at a time.