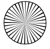
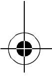
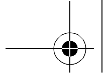
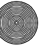



Chapter 2

Principles

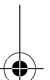
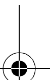
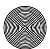
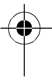
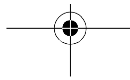
Principles and Practices

Principles are underlying truths that don't change over time or space, while practices are the application of principles to a particular situation. Practices can and should differ as you move from one environment to the next, and they also change as a situation evolves.

Let's say that you want to change your software development practices because they aren't working very well, but it's not clear what new practices you should adopt. In other words, you would like to move away from something, but it's not clear where you are heading. As you look for a new approach to software development, is it better to spend your time understanding principles or studying practices?

There is a learn-by-doing approach: Adopt a coherent set of practices with confidence that this will eventually lead to understanding of the principles behind them. Then there is an understand-before-doing approach: Understand the underlying principles, and use them to develop practices for specific situations. We observe that the best results come from combining the two approaches. Copying practices without understanding the underlying principles has a long history of mediocre results. But when the underlying principles are understood, it is useful to copy practices that work for similar organizations and modify them to fit your environment. This can provide a jump-start to implementing the principles.

For example, when Mary's video tape plant implemented Just-in-Time, it was not possible to simply copy Toyota's practices because her plant was a process plant, not an assembly plant. The management team had to think carefully about what Just-in-Time meant in their world. They decided to use the Toyota Kanban system and implement pull scheduling, which were specific practices. But they knew that inventory reduction alone was not enough to achieve breakthrough cost reduction, so they developed unique approaches to engage pro-

ch02.fm Page 20 Thursday, August 3, 2006 9:22 AM

duction workers and create a stop-the-line culture. This combination of understanding principles and adapting practices led to dramatic success.

Many companies copied Toyota's Kanban system during the 1990s with mediocre results. Those companies didn't see lean as a management system aimed at eliminating waste. We suspect that many companies with Kanban systems left plenty of waste in their value streams. Similarly, many companies will implement some of the agile software development practices discussed later in this book, but if they don't recognize waste and manage the value stream so as to eliminate it, if they don't possess a deep respect for their workers and partners, results may be similarly mediocre.

Software Development

Lean practices from manufacturing and supply chain management don't translate easily to software development, because both software and development are individually quite different than operations and logistics. Let's take a look at each of these words—software and development—and analyze just where their uniqueness lies.

Software

Embedded software is the part of a product that is expected to change. If it didn't need to change, then it might as well be hardware. Enterprise software is the part of a business process that bears the brunt of its complexity. If there is a messy calculation or complicated information flow, software gets assigned the job. Software gets the user interaction jobs, the last minute jobs, the "figure it out in the future" jobs. Because of this, almost everything we know about good software architecture has to do with making software easy to change.[1] And that's not a surprise because well over half of all software is developed after first release to production.[2]

1. For example, in the article "Quality With a Name," Jim Shore defines a high-quality software architecture this way: "A good software design minimizes the time required to create, modify, and maintain the software while achieving acceptable run-time performance." See: www.jamesshore.com/Articles/Quality-With-a-Name.html
2. The percentage of software lifecycle cost attributed to "maintenance" ranges between 40 percent and 90 percent. See Kajko-Mattsson, Mira, Ulf Westblom, Stefan Forssander, Gunnar Andersson, Mats Medin, Sari Ebarasi, Tord Fahlgren, Sven-Erik Johansson, Stefan Törnquist, and Margareta Holmgren, "Taxonomy of Problem Management Activities." Proceedings of the Fifth European Conference on Software Maintenance and Reengineering, March 2001, 1–10.

As time goes on, modifying production software tends to get more difficult and expensive. Changes add complexity, and complexity usually calcifies the code base, making it brittle and easy to break. Far too often companies are startled to discover that their software investment has turned into a tangle of unmanageable code. But this doesn't have to happen. The best software products have been around for a decade or more, and every useful product of that age has been changed regularly over its lifetime. These products have architectures and development processes that build change tolerance into the code. All code worthy of the name software should be designed and built with change tolerance in mind.

Development

Development is the process of transforming ideas into products. There are two schools of thought about how go about this transformation: We might call one the deterministic school of thought and the second the empirical school of thought. The deterministic school starts by creating a complete product definition, and then creates a realization of that definition. The empirical school starts with a high-level product concept and then establishes well-defined feedback loops that adjust activities so as to create an optimal interpretation of the concept.

The Toyota Product Development System sits squarely in the empirical school, starting with a vehicle concept rather than a definition and empirically fleshing out the concept into a product throughout the development process. For example, the product concept of the Prius did not mention a hybrid engine; it set a fuel economy target of 20 kilometers per liter (47.5 miles per gallon). The Prius product concept also called for a roomy cabin for passengers, but did not set vehicle dimensions. Only during development did the team settle on the hybrid engine, barely out of the research lab, as the best way to achieve the aggressive fuel economy target.[3]

We believe that any development process that deals with a changing environment should be an empirical process, because it provides the best known approach for adapting to change. As we noted above, software by its very nature should be designed to adapt to change both during initial development and over its lifecycle. Thus, software development should be an empirical process.

3. Jeffrey Liker, *The Toyota Way*, McGraw-Hill, 2004. See Chapter 6 on the development of the Prius.

What Is This Thing Called "Waterfall?"

I didn't run into the term "waterfall" until 1999 when I started working on a government project. I had been a (very good) programmer in the 1970s, writing software that controlled machines. This might be called embedded software today, but in those days the computers were hardly small enough to be embedded. I worked on some large projects that involved building and starting up complex process lines to make tape.[4] My colleagues on these projects were engineers with years of experience developing large, complex tape-making lines. The projects were managed by seasoned experts, who knew how to get approval for an overall budget and schedule, then turn things over to the experts to adapt the plan as needed to deliver a working line. Everyone was well aware of the fact that while budget and schedule were important, the overriding goal was to deliver a line that made excellent product. And we always did.

When I became the information systems manager of a video cassette plant, I used the approach I had learned as an engineer to manage the development of new software, always keeping in mind that my department's primary job was to support production. Later I moved into product development, where we used a rigorous but very adaptable stage-gate process to commercialize new products.

Thus, I managed to escape working with the waterfall methodology until I bumped into it in 1999 on that government project. I was puzzled by the waterfall approach because I couldn't understand how it could possibly work; and as a matter of fact, it didn't work. As I compared my experience working on complex, successful projects to the prescribed waterfall approach which failed on a relatively small project, I decided to write a book about what really works.[5] In that book we outlined seven principles of software development, which are summarized below.

—Mary Poppendieck

4. The equipment needed for making tape bears some resemblance to the automated looms manufactured by Toyoda Automatic Loom in the 1920s.
5. Mary and Tom Poppendieck, *Lean Software Development: An Agile Toolkit*, Addison-Wesley, 2003.

Poppendieck.book Page 23 Thursday, July 27, 2006 11:46 AM

The Seven Principles of Lean Software Development

In this section we summarize the seven principles of software development that formed the core of our previous book. Some readers may notice that the wording of some principles has changed a bit, but the intent remains the same. After each principle we add a prevailing myth, which makes the principle counterintuitive for those who believe the myth.

Principle 1: Eliminate Waste

Taiichi Ohno called the Toyota Production System a management system for "the absolute elimination of waste."[6] When asked how it worked, he said, "All we are doing is looking at the timeline from the moment a customer gives us an order to the point when we collect the cash. And we are reducing that timeline by removing the nonvalue-added wastes."[7] In a nutshell, that's what Lean Production is all about. In Lean Software Development the goal of eliminating waste is the same, but the start and end of the timeline may be modified. You start the timeline clock when you receive an order to address a customer need (whatever that means in your organization) and stop it when software is deployed that addresses the need. Lean Software Development focuses on reducing that timeline by removing all nonvalue-adding wastes.

To eliminate waste, you first have to recognize it. Since waste is anything that does not add value, the first step to eliminating waste is to develop a keen sense of what value really is. There is no substitute for developing a deep understanding of what customers will actually value once they start using the software. In our industry, value has a habit of changing because, quite often, customers don't really know what they want. In addition, once they see new software in action, their idea of what they want will invariably shift. Nevertheless, great software development organizations develop a deep sense of customer value and continually delight their customers. Think of Google; it regularly and repeatedly delights customers around the world.

Once you have a good understanding of value, the next step is to develop a capability to really see waste. Waste is anything that interferes with giving customers what they value at the time and place where it will provide the most value. Anything we do that does not add customer value is waste, and any delay that keeps customers from getting value when they want it is also waste.

---

6. Taiichi Ohno, *Toyota Production System: Beyond Large Scale Production*, Productivity Press, 1988, p. 4.
7. Ibid., p. ix.

In manufacturing, inventory is waste. Inventory has to be handled, moved, stored, tracked, and retrieved. Not only does this take time and effort, but it also adds complexity—a large cost multiplier. Inventory gets lost, grows obsolete, hides quality problems, and ties up money. So one goal in manufacturing is to carry as little inventory as possible.

The inventory of software development is partially done work. Partially done software has all of the evils of manufacturing inventory: It gets lost, grows obsolete, hides quality problems, and ties up money. Moreover, much of the risk of software development lies in partially done work.

A big form of waste in software development is “churn.” In our classes, we’ve often encountered “requirements churn” in the 30 percent to 50 percent range, and we’ve seen many test-and-fix cycles that take twice as long as the initial development time. We’ve found that software development churn is always associated with large inventories of partially done work. When requirements are specified long before coding, of course they change. When testing occurs long after coding, test-and-fix churn is inevitable. Unfortunately, these types of churn are often just a precursor to the ever larger churn created by delayed (aka big-bang) integration.

But far and away the biggest source of waste in software development is extra features. Only about 20 percent of the features and functions in typical custom software are used regularly. Something like two-thirds of the features and functions in typical custom software are rarely used.[8] We are not talking about the necessary safety or security features. We are talking about features that really weren’t needed in the first place. There is a *huge* cost to developing extra features in a software system. They add complexity to the code base that will drive up its cost at an alarming rate, making it more and more expensive to maintain, dramatically reducing its useful life.

Myth: Early Specification Reduces Waste

The reason we develop all of this extra code lies in a game we play with our customers. We set up the rules:

> Will you, valued customer, please give us a list of everything you want the software to do? We will write it down and ask you to sign off on the result. After that, if you want changes or additions, you will have to go through an arduous process

8. When Jim Johnson, chairman of the Standish Group reported these ratios at XP 2002 in Sardinia, they came from a limited study. Since then we have asked almost every group we address if these numbers match their experience, particularly if they are developing custom software. The response has always confirmed that these numbers, if not precisely correct, are in the right ballpark.

Poppendieck.book Page 25 Thursday, July 27, 2006 11:46 AM

called "change management" to get a change approved. So you had better think of everything you want right now, because we have to know about all of it at the beginning in order to develop good software.

Is it any wonder that our customers throw everything, including the kitchen sink, into their list of requirements? Far too often the game of scope control has the opposite effect—it creates scope bloat. Just as Taiichi Ohno called overproduction the worst waste in manufacturing, unused features are the worst kind of waste in software development. Every bit of code that is there and not needed creates complexity that will plague the code base for the rest of its life. Unused code still requires unnecessary testing, documentation, and support. It will do its share of making the code base brittle and difficult to understand and change as time goes on. The cost of complexity in code dominates all other costs, and extra features that turn out to be unnecessary are one of the biggest killers of software productivity.

We need a process that allows us to develop the 20 percent of the code that will deliver 80 percent of the value, and only then go on to develop the next most important features. We should never establish scope with a list of everything that a system might possibly need, especially if the list comes from customers who don't really know what they want.

Principle 2: Build Quality In

"We need more discipline around here, not less," skeptical managers often tell us. Lean Software Development is very disciplined, we always respond. It's just that you look at discipline a bit differently. Your goal is to build quality into the code from the start, not test it in later. You don't focus on putting defects into a tracking system; you avoid creating defects in the first place. It takes a highly disciplined organization to do that.

Defect Queues

We were visiting an organization that was certified at CMM Level 4 and on the verge of reaching CMM Level 5. We were impressed with the people and the discipline; we were not surprised that their products were doing very well.

But something bothered us. Recently their release cycle time had slowed down from six weeks to four months, and they asked us to help them figure out why. We sketched the release cycle timeline on a board, and at the end of the cycle we found a four-week testing period. "It's no wonder you can't release in six weeks. You take four weeks to test your product!" I said.

"I know. We should automate testing," the quality manager replied, "but that seems rather simplistic." I agreed. This was a top-notch organization. There was probably more to this problem than automated testing would resolve.

"Let's do a Pareto analysis[9] of the four weeks of testing," I suggested. "What do you do during the four weeks that takes the most time?"

"Fix the defects," was the immediate response.

"Oh, so you're not really testing during the four weeks, you're fixing defects!" I said. "It's not fair to call that testing. What if there were no defects to fix—*then* how long would testing take?"

"Oh, maybe two or three days," came the answer.

"So, if you could do testing in two or three days, *then* could you move your releases back to every six weeks?" I asked.

"Definitely," was the immediate reply.

"Well, it seems that you are testing too late in your process. You really should be finding these defects much sooner," I suggested.

"Oh, but we are!" he said. "They're in our defect tracking system."

"You mean you know about these defects, but you're not doing anything about them until the end?" I asked, astonished.

"Yes," came the sheepish reply.

"OK, you know what you have to do," I said. "Don't record the defects when you find them, just fix them! Get your final testing down to a couple of days by setting the expectation that the code should work when final verification begins." I had no doubt that this very competent organization would be able to meet this goal once the team decided that it was important.

—Mary Poppendieck

9. A Pareto analysis is an application of the "vital few and trivial many" rule, also called the "80/20" rule, first popularized by quality guru J.M. Juran. The analysis proceeds thus: Divide a problem into categories, find the biggest category, look for the root cause of the problem creating that category, and fix it. Once the cause of the biggest problem is eliminated, it's time to repeat the cycle with another Pareto analysis on the remaining problems. Chapter 7 contains a more detailed example of a team using this technique.

Poppendieck.book Page 27 Thursday, July 27, 2006 11:46 AM

According to Shigeo Shingo, there are two kinds of inspection: inspection after defects occur and inspection to prevent defects.[10] If you really want quality, you don't inspect after the fact, you control conditions so as not to allow defects in the first place. If this is not possible, then you inspect the product after each small step, so that defects are caught immediately after they occur. When a defect is found, you stop-the-line, find its cause, and fix it immediately.

Defect tracking systems are queues of partially done work, queues of rework if you will. Too often we think that just because a defect is in a queue, it's OK, we won't lose track of it. But in the lean paradigm, queues are collection points for waste. The goal is to have no defects in the queue, in fact, the ultimate goal is to eliminate the defect tracking queue altogether. If you find this impossible to imagine, consider Nancy Van Schooenderwoert's experience on a three-year project that developed complex and often-changing embedded software.[11] Over the three-year period there were a total of 51 defects after unit testing with a maximum of two defects open at once. Who needs a defect tracking system for two defects?

Today we have the tools to keep most defects out of code as it is being developed. A group of developers using test-driven-development writes unit tests and acceptance tests before they write the associated code. They integrate both code and tests into the system as often as possible—every hour or so—and run a test harness to see that no defects have been introduced. If the tests don't pass, they do not add new code until the problem is fixed or the failing code is backed out. At the end of the day, a longer and more complete test harness is run. Every weekend the system is attached to an even more comprehensive test harness. Organizations using this process report incredibly low defect rates, as well as very short timeframes to find the cause of defects that are discovered.

10. See Shigeo Shingo, *Study of 'Toyota' Production System*, Productivity Press, 1981, Chapter 2.3.
11. Nancy Van Schooenderwoert and Ron Morsicato, "Taming the Embedded Tiger—Agile Test Techniques for Embedded Software," Proceedings, Agile Development Conference, Salt Lake City, June, 2004, and Nancy Van Schooenderwoert, "Embedded Agile Project by the Numbers with Newbies," Proceedings, Agile 2006 Conference, Minneapolis, July 2006.

Poppendieck.book Page 28 Thursday, July 27, 2006 11:46 AM

Productivity Soared[12]

Test-driven development (TDD) is an amazingly effective approach to improving code quality. In February 2004 I introduced the technique to a company that was struggling with quality issues. Although none of the code (which was written in Java) was more than five years old, they already considered this to be a legacy application. On average there were ten defects reported per every thousand lines of noncomment source statements (NCSS) in the six months following release.

The poor quality of the company's software was affecting its reputation, customer satisfaction, and its ability to add new features in a timely manner.

To correct these problems the team began using the Scrum[13] agile development process and soon afterwards adopted TDD. Following the adoption of TDD the number of defects reported dropped to less than three per thousand lines of code. Even this 70 percent reduction in defects understates the improvement because the team didn't track whether a reported defect was caused by code written before or after TDD. That is, some of the defects being reported were still the result of code written before TDD was adopted. Conservatively, the team probably achieved an 80 percent to 90 percent improvement.

The improved quality paid dividends in more than one way. Because the team spent so much less time tracking down bugs (both in the debugger during development and afterward when a user reported a bug) productivity soared. Prior to adopting TDD the team completed an average of seven function points per person-month. After adopting TDD they completed an average of 23 function points per person-month. The team was over three times more productive with far fewer defects.

—Mike Cohn, President, Mountain Goat Software

Myth: The Job of Testing Is to Find Defects

The job of tests, and the people that develop and runs tests, is to *prevent* defects, not to find them. A quality assurance organization should champion processes that build quality into the code from the start rather than test quality

12. Thanks to Mike Cohn for sharing his experience. Used with permission.
13. Scrum is a well-known iterative development methodology. See Ken Schwaber and Mike Beedle, *Agile Software Development with SCRUM*, Prentice Hall, 2001, and Ken Schwaber, *Agile Project Management with Scrum*, Microsoft, 2004.

Poppendieck.book Page 29 Thursday, July 27, 2006 11:46 AM

in later. This is not to say that verification is unnecessary. Final verification is a good idea. It's just that finding defects should be the exception, not the rule, during verification. If verification routinely triggers test-and-fix cycles, then the development process is defective.

In Mary's plant, common wisdom held that 80 percent of the defects that seemed to be caused by people making mistakes were actually caused by a system that allowed the mistakes to happen. Therefore, the vast majority of defects were actually management problems. The slogan, "Do it right the first time," was the rallying cry for doing away with after-the-fact inspection. It meant mistake-proofing each manufacturing step so that it was easy to avoid creating defects.

When we use the slogan "Do it right the first time," in software development, it should mean that we use test-driven development and continuous integration to be sure the code behaves exactly as intended at that point in time. Unfortunately, the slogan has been used to imply something very different. "Do it right the first time," has been interpreted to mean that once code is written, it should never have to be changed. This interpretation encourages developers to use some of the worst known practices for the design and development of complex systems. It is a dangerous myth to think that software should not have to be changed once it is written.

Let's revisit the best opportunity for eliminating waste in software development: *Write less code*. In order to write less code, we need to find the 20 percent of the code that will provide 80 percent of the value and write that first. Then we add more features, stopping when the value of the next feature set is less than its cost. When we add new features, we have to keep the code simple and clean, or complexity will soon overwhelm us. So we refactor the design to take the new capability into account, keep the code base simple, and get rid of software's worst enemy: duplication. This means that we should routinely expect to change existing code.

Principle 3: Create Knowledge

One of the puzzling aspects of "waterfall" development is the idea that knowledge, in the form of "requirements," exists prior to and separate from coding. Software development is a knowledge-creating process. While an overall architectural concept will be sketched out prior to coding, the validation of that architecture comes as the code is being written. In practice, the detailed design of software always occurs during coding, even if a detailed design document was written ahead of time. An early design cannot fully anticipate the complexity encountered during implementation, nor can it take into account the ongoing

Poppendieck.book Page 30 Thursday, July 27, 2006 11:46 AM

feedback that comes from actually building the software. Worse, early detailed designs are not amenable to feedback from stakeholders and customers. A development process focused on creating knowledge will expect the design to evolve during coding and will not waste time locking it down prematurely.

Alan MacCormack, a professor at Harvard Business School, spends his time studying how organizations learn. A while back, when he was studying software development practices, a company asked him to evaluate two of its projects—a "good" project and a "bad" project.[14] The "good" project was run by-the-book. It had an optimized design, and the resulting system was very close to the initial specification. The "bad" project underwent constant change as the development team struggled to understand and respond to changes in the market.

When the projects were evaluated using MacCormack's criteria, the "good" project scored poorly in quality, productivity, and market acceptance while the "bad" project was a market success. It's not surprising that the team that learned from the market throughout development created a better product. The real eye-opener was that the company's managers thought that learning about the market and building a product to satisfy it was "bad."

MacCormack has identified four practices that lead to successful software development:[15]

1. Early release of a minimum feature set to customers for evaluation and feedback
2. Daily builds and rapid feedback from integration tests
3. A team and/or leader with the experience and instincts to make good decisions
4. A modular architecture that supports the ability to easily add new features

Companies that have exhibited long-term excellence in product development share a common trait: They generate new knowledge through disciplined experimentation and codify that knowledge concisely to make it accessible to the

---

14. This story is told in "Creating a Fast and Flexible Process: Research Suggests Keys to Success" by Alan MacCormack, at www.roundtable.com/MRTIndex/FFPD/ART-maccormack.html and at www.agiledevelopmentconference.com/ 2003/files/AlanAgileSoftwareJun03.ppt.
15. In addition to the above article, see also Alan MacCormack, "Product-Development Practices That Work: How Internet Companies Build Software," *MIT Sloan Management Review*, Winter 2001, Vol. 40 number 2.

Poppendieck.book Page 31 Thursday, July 27, 2006 11:46 AM

larger organization. Not only do these companies capture explicit data, they find ways to make tacit knowledge explicit and make it part of the organizational knowledge base.[16] These companies realize that while learning about the product under development is important, codifying the knowledge for use in future products is essential.

It is important to have a development process that encourages systematic learning throughout the development cycle, but we also need to systematically improve that development process. Sometimes in the search for "standard" processes we have locked our processes up in documentation that makes it difficult for development teams to continually improve their own processes. A lean organization knows that it must constantly improve its processes because in a complex environment there will always be problems. Every abnormality should trigger a search for the root cause, experiments to find the best way to remedy the problem, and a change in process to keep it from resurfacing. Process improvement efforts should be the responsibility of development teams, and every team should set aside time to work on process improvement on a regular basis.

Myth: Predictions Create Predictability

Predictable outcomes are one of the key expectations that the marketplace imposes on companies and their senior management, and these expectations eventually flow down to software development. Unfortunately, software development has a notorious reputation for being unpredictable, so there is a great deal of pressure to make it more predictable. The paradox is that in our zeal to improve the predictability of software development, we have institutionalized practices that have had the opposite effect. We create a plan, and then we act on that plan as if it embodies an accurate prediction of the future. Because we assume that our predictions are fact, we tend to make early decisions that lock us into a course of action that is difficult to change. Thus, we lose our capability to respond to change when our predictions turn out to be inaccurate. The solution to this problem, it would seem, is to make more accurate predictions.

We forget that the predictions of the future are always going to be inaccurate if they are 1) complex, 2) detailed, 3) about the distant future, or 4) about an uncertain environment. No amount of trying to make these kinds of predictions more accurate is going to do much good. There are, however, well-proven ways to create reliable outcomes even if we cannot start with accurate predictions.

16. See Ikujiro Nonaka and Hirotaka Takeuchi in *The Knowledge Creating Company: How Japanese Companies Create the Dynamics of Innovation*, Oxford University Press, 1995, p. 225.

Poppendieck.book Page 32 Thursday, July 27, 2006 11:46 AM

The idea is to stop acting as if our predictions of the future are fact rather than forecast. Instead, we need to reduce our response time so we can respond correctly to events as they unfold.

In order to increase the predictability of outcomes, we need to decrease the amount of speculation that goes into making decisions. Decisions that are based on facts, rather than forecasts, produce the most predictable results. As a control engineer, Mary knows that an empirical control system—one based on feedback—delivers more predictable results than a deterministic control system. Fundamentally, an organization that has a well-developed ability to wait for events to occur and then respond quickly and correctly will deliver far more predictable outcomes than an organization that attempts to predict the future.

Principle 4: Defer Commitment

Emergency responders are trained to deal with challenging, unpredictable, and often dangerous situations. They are taught to assess a challenging situation and decide how long they can wait before they must make critical decisions. Having set a timebox for such a decision, they learn to wait until the end of the timebox before they commit to action, because that is when they will have the most information.

We should apply the same logic to the irreversible decisions that need to be made during the course of software development: Schedule irreversible decisions for the last responsible moment, that is, the last chance to make the decision before it is too late. This is not to say that all decisions should be deferred. First and foremost, we should try to make most decisions reversible, so they can be made and then easily changed. One of the more useful goals of iterative development is to move from "analysis paralysis" to getting something concrete accomplished. But while we are developing the early features of a system, we should avoid making decisions that will lock in a critical design decision that will be difficult to change. A software system doesn't need complete flexibility, but it does need to maintain options at the points where change is likely to occur. A team and/or leader with experience in the domain and the technology will have developed good instincts and will know where to maintain options.

Many people like to get tough decisions out of the way, to address risks head-on, to reduce the number of unknowns. However, in the face of uncertainty especially when it is accompanied by complexity, the more successful approach is to tackle tough problems by experimenting with various solutions, leaving critical options open until a decision must be made. In fact, many of the best software design strategies are specifically aimed at leaving options open so that irreversible decisions can be made as late as possible.

Poppendieck.book Page 33 Thursday, July 27, 2006 11:46 AM

Myth: Planning Is Commitment

"In preparing for battle I have always found that plans are useless, but planning is indispensable." Dwight Eisenhower's famous quote gives us good perspective on the difference between planning and commitment. Planning is an important learning exercise, it is critical in developing the right reflexes in an organization, and it is necessary for establishing the high-level architectural design of a complex system.

Plans, on the other hand, are overrated. Listen to Taiichi Ohno:[17]

> Plans change very easily. Worldly affairs do not always go according to plan and orders have to change rapidly in response to change in circumstances. If one sticks to the idea that once set, a plan should not be changed, a business cannot exist for long.
>
> It is said that the sturdier the human spine, the more easily it bends. This elasticity is important. If something goes wrong and the backbone is placed in a cast, this vital area gets stiff and stops functioning. Sticking to a plan once it is set up is like putting the human body in a cast. It is not healthy.

Barry Boehm and Richard Turner[18] coined the term "plan-driven methods" to describe software development approaches that are based on the expectation that a plan is a commitment. They define a capable plan-driven process as one that has "the inherent capability … to produce planned results."[19] They further note that plan-driven methods originated in the government contracting world.[20] Indeed, it is a challenge to administer contracts in the arms-length manner required by government organizations without creating a detailed plan that is regarded as a commitment.

But in the commercial world, wise businesses (such as Toyota) realize that sticking to a detailed plan is not healthy, and measuring process capability against ones ability to do so is measuring the wrong thing. Let's not succumb to the myth that planning is the same thing as making a commitment. We should plan thoughtfully and commit sparingly.

17. Taiichi Ohno, *Toyota Production System: Beyond Large Scale Production*, Productivity Press, 1988, p. 46.
18. See Barry Boehm and Richard Turner, *Balancing Agility and Discipline: A Guide for the Perplexed*, Addison-Wesley, 2004.
19. Ibid., p. 12.
20. Ibid., p. 10.

Poppendieck.book Page 34 Thursday, July 27, 2006 11:46 AM

Principle 5: Deliver Fast

When our children were so little that their chins were at table level when they sat in a chair, we could always find a couple of thick Sears catalogs in any house we visited and use them as a booster seat. Everyone loved to page through the catalogs, and we often ordered from them. Delivery took two or three weeks, so we didn't bother to order things we could get in local stores. Somewhere in the middle of the 1980s, a mail order company in Maine called L.L. Bean decided to compete with Sears on the basis of time. Its goal was to ship every order within 24 hours after the order arrived. This was such an amazing concept that other companies used to tour the L.L. Bean distribution center to see how the company did it.

It didn't take long for two- to three-week delivery times to seem awfully slow. The venerable Sears catalog, approaching its 100th birthday, was unable to compete and closed up shop in 1993. L.L. Bean had much lower costs, partly because it didn't have to incur the expense of tracking unfilled orders. Think about it: If we ordered a yellow shirt from Sears, we could call up a week or two later and say, "Guess what, I've changed my mind. Can you make that a blue shirt?" And they would change the order. But if we called L.L. Bean with the same request, they would say, "Have you looked on your doorstep? It should be arriving today."

The moral of this story is that we need to figure out how to deliver software so fast that our customers don't have time to change their minds.

FedEx pretty much invented overnight shipping. Southwest pioneered rapid turnaround at airline gates. Both companies have maintained leadership in their respective industry even after widespread imitation. Dell created a speed-based assemble-to-order model that has proven remarkably difficult to imitate. Toyota was able to bring the revolutionary hybrid Prius to market in 15 months, a product development speed that few automobile companies can begin to approach.

Companies that compete on the basis of time often have a significant cost advantage over their competitors: They have eliminated a huge amount of waste, and waste costs money. In addition, they have extremely low defect rates. Repeatable and reliable speed is impossible without superb quality. Furthermore they develop a deep customer understanding. They are so fast that they can afford take an experimental approach to product development, trying new ideas and learning what works.

Poppendieck.book Page 35 Thursday, July 27, 2006 11:46 AM

Myth: Haste Makes Waste

In the software development industry, it has long been thought that in order to get high quality you have to, "slow down and be careful." But when an industry imposes a compromise like that on its customers, the company that breaks the compromise stands to gain a significant competitive advantage.[21] Google and PatientKeeper, featured later in this book, are just two of many companies that deliver software very fast and with high quality. Software development organizations that continue to believe speed and quality are incompatible face the prospect of going the way of the Sears catalog in the not-too-distant future.

Caution: Don't equate high speed with hacking. They are worlds apart. A fast-moving development team must have excellent reflexes and a disciplined, stop-the-line culture. The reason for this is clear: You can't sustain high speed unless you build quality in.

In a quest for discipline, many organizations develop detailed process plans, standardized work products, workflow documentation, and specific job descriptions. This is often done by a staff group that trains workers in the process and monitors conformance. The goal is to achieve a standardized, repeatable process which, among other things, makes it easy to move people between projects.[22] But a process designed to create interchangeable people will not produce the kind of people that are essential to make fast, flexible processes work.

If you want to go fast, you need engaged, thinking people who can be trusted to make good decisions and help each other out. In fast-moving organizations, the work is structured so that the people doing the work know what to do without being told and are expected to solve problems and adapt to changes without permission. Lean organizations work to standards, but these standards exist because they embody the best current knowledge about how to do a job. They form a baseline against which workers are expected to experiment to find better ways to do their job. Lean standards exist to be challenged and improved.[23]

There are two ways to achieve high quality. You can slow down and be careful, or you can develop people who continually improve their processes, build quality into their product, and develop the capability to repeatedly and reliably respond to their customers many times faster than their competitors.

---

21. George Stalk and Rob Lachenauer, *Hardball: Are You Playing to Play or Playing to Win*, Harvard Business School Press, 2004.
22. See Barry Boehm and Richard Turner, *Balancing Agility and Discipline: A Guide for the Perplexed*, Addison-Wesley, 2004, pp. 11–12.
23. See Jim Shook "Bringing the Toyota Production System to the United States" in *Becoming Lean*, Jeffrey Liker, editor, Productivity Press, 2004, pp. 59–60.

Principle 6: Respect People

When Joel Spolsky was fresh out of college and a new employee at Microsoft, he was tasked with developing the macro language strategy for Excel. He spent some time learning what customers might want macros to do and wrote a spec. Much to his surprise, an applications architecture group got wind of the spec and asked to review it. Joel thought they might have some good advice for him, but he found that the small group knew even less about macros than he did—even with four Ph.D.'s and a high-profile boss (a friend of Bill Gates, who was something like employee number 6). Despite a superficial idea of how customers might use macros, the app architecture group thought it was their job to determine how macros should be implemented. Joel's managers made it clear that the decisions were his, and his programming team backed him up. But the app architecture group wasn't happy about that, and their boss called a big meeting to complain about how Joel was messing up the macro strategy.

If Microsoft were an ordinary company, you can imagine how this scenario might play out: The app architecture group gets its way, and Joel either acquiesces or leaves. But that is not what happened. The day after the big meeting, a senior vice president checked in with Joel in the cafeteria and asked, "How's it going with the app architecture group?" Joel said everything was fine, of course. But the next day Joel heard via the grapevine that the app architecture group had been disbanded. He was impressed. "At Microsoft," says Joel, "if you're the Program working on the Excel macro strategy, even if you've been at the company for less than six months, it doesn't matter—you are the *GOD* of the Excel macro strategy, and nobody, not even employee number 6, is allowed to get in your way. Period." [24]

Joel's story shows what Piniciple 6: Respect People means in software development from the point of view of the people doing the work.

It's noteworthy that three of the four cornerstones of the Toyota Product Development System in Figure 1.5[25] concern the people involved in the product development process. Looking at these three cornerstones gives us a broader idea of what respecting people means:

24. Joel Spolsky, "Two Stories, " http://www.joelonsoftware.com/articles/twostories.html. Used with permission.
25. Michael Kennedy, *Product Development for the Lean Enterprise: Why Toyota's System Is Four Times More Productive and How You Can Implement It*, Oaklea Press, 2003, p. 120. Used with permission.

ch02.fm Page 37 Thursday, August 3, 2006 9:23 AM

1. **Entrepreneurial Leader:** People like to work on successful products, and highly successful products can usually be traced to excellent leaders. A company that respects its people develops good leaders and makes sure that teams have the kind of leadership that fosters engaged, thinking people and focuses their efforts on creating a great product.

2. **Expert Technical Workforce:** Any company that expects to maintain a competitive advantage in a specific area must develop and nurture technical expertise in that area. Companies that buy all the expertise they need will find that their competitors can buy it also. Companies that see no need for expertise will find that they have no sustainable competitive advantage. Wise companies make sure that appropriate technical expertise is nurtured and teams are staffed with the needed expertise to accomplish their goals.

3. **Responsibility-Based Planning and Control:** Respecting people means that teams are given general plans and reasonable goals and are trusted to self-organize to meet the goals. Respect means that instead of telling people what to do and how to do it, you develop a reflexive organization where people use their heads and figure this out for themselves.

Myth: There Is One Best Way

The book *Cheaper by the Dozen*[26] is a very funny, true story about a family of 12 children whose father is an efficiency expert. Since Mary grew up the fourth child in a family of 11 children, she eagerly read this book as a child. What escaped Mary until she re-read the book recently is that *Cheaper by the Dozen* gives us a unique perspective into the origins of scientific management. The father of the dozen, Frank Gilbreth, was a world-renowned consultant who insisted that there is always "one best way" to do everything. Reading between the lines, you can guess from Gilbreth's regimented household that most workers would not want to be on the receiving end of one of his efficiency efforts. But if Frank Gilbreth had a humorous lack of respect for people, then his colleague Frederick Winslow Taylor exhibited a barely masked distain for "laborers" in his writings.

After Frank Gilbreth's early death, his wife and partner, Lillian Gilbreth, took over the business and became one of the foremost industrial engineers of

26. Frank B. Gilbreth Jr. and Ernestine Gilbreth Carey, *Cheaper by the Dozen*, T.Y. Crowell Co., 1948. The movie by the same name is quite a departure from the book.

ch02.fm Page 38 Thursday, August 3, 2006 9:23 AM

her time. She moved away from the idea of "one best way" both in raising her family[27] and in her professional work, where she focused on worker motivation.

But the damage was done. The task of finding and institutionalizing the "one best way" to do every job was handed over to industrial engineers in most American industries. Forerunners of the so-called process police, these industrial engineers often created standards without really understanding the work and enforced them without entertaining the possibility that there may be a better way.

There is no such thing as "one best way." There is no process that cannot be improved. To prove this to yourself, simply spend some time quietly watching people doing their work. After a while you will notice many things that could be improved. Processes should be improved by the work team doing the job. They need the time, the charter, and the guidance to tackle their problems, one at a time, biggest problem first. This never-ending continuous improvement process should be found in every software development organization.

Principle 7: Optimize the Whole

Software development is legendary for its tendency to suboptimize.

- **Vicious Circle No. 1 (of course, this would never happen at your company):**
  - A customer wants some new features, "yesterday."
  - Developers hear: Get it done fast, at all costs!
  - Result: Sloppy changes are made to the code base.
  - Result: Complexity of the code base increases.
  - Result: Number of defects in the code base increases.
  - Result: There is an exponential increase in time to add features.
- **Vicious Circle No. 2 (and this wouldn't happen at your company either):**
  - Testing is overloaded with work.
  - Result: Testing occurs long after coding.
  - Result: Developers don't get immediate feedback.
  - Result: Developers create more defects.

27. See the sequel, Frank B. Gilbreth Jr. and Ernestine Gilbreth Carey, *Belles on Their Toes*, T.Y. Crowell Co., 1950.

Poppendieck.book Page 39 Thursday, July 27, 2006 11:46 AM

- Result: Testing has more work. Systems have more defects.
- Result: Feedback to developers is delayed further. Repeat cycle.

A lean organization optimizes the whole value stream, from the time it receives an order to address a customer need until software is deployed and the need is addressed. If an organization focuses on optimizing something less than the entire value stream, we can just about guarantee that the overall value stream will suffer. We have seen this many times in value stream maps from our classes: Almost every time we spot a big delay, there is a handoff of responsibility from one department to the next, with no one responsible for shepherding the customers' concerns across the gap.

Organizations usually think that they are optimizing the whole; after all, everyone knows that suboptimization is bad, so no one wants to admit to it. Yet in surprisingly many cases, suboptimization is institutionalized in the measurement system. Consider the case of Fujitsu.[28] In 2001, Fujitsu took over the help desk of BMI, a UK airline. It analyzed the calls it received from BMI employees and found that 26 percent of the calls were for malfunctioning printers at airline check-in counters. Fujitsu measured how long the printers were down and totaled up the cost to BMI. Then Fujitsu prepared a business case and convinced BMI management to replace the printers with more robust machines. Fujitsu continued to uncover and attack the root causes of calls to the help desk, so that after 18 months, total calls were down 40 percent.

This seems like a good story, but there is something wrong. Most help desks get paid based on the number of calls handled. By reducing calls 40 percent, Fujitsu was reducing its revenue by the same amount! As it happens, Fujitsu's results were so dramatic that it was able to renegotiate its revenue agreement with BMI so that it got paid on "potential" calls rather than actual calls. Under the new agreement, Fujitsu had a financial incentive to continue to really help BMI solve its business problems.

This story points out that outsourced call centers working under the traditional revenue model have no incentive to find and resolve the causes of customer problems.

Crossing organizational boundaries is expensive. Peter Drucker noted that a company that maintains a single management system throughout the value stream will see a 25 percent to 30 percent cost advantage over competitors.[29]

---

28. James P. Womack and Daniel T. Jones, *Lean Consumption: How Companies and Customers Can Create Value and Wealth Together*, Free Press, 2005, pp. 58–63.
29. Peter Drucker, *Management Challenges for the 21st Century*, Harper Business, 1999, p. 33.

Poppendieck.book Page 40 Thursday, July 27, 2006 11:46 AM

Thus, there's a sizable amount of savings to be gained from structuring contracts, outsourcing agreements, and cross-functional interactions with correct incentives that make sure everyone is focused on optimizing the whole.

Myth: Optimize By Decomposition

Alfred P. Sloan invented an organizational structure designed to deal with complexity: He created decentralized divisions and used financial metrics to judge the managers' performance. This was a great improvement over Ford's hands-on control; it helped General Motors produce a variety of cars and overtake Ford as the market leader. Ever since, companies have been working to find the right metrics to manage performance.

While Sloan's use of metrics to manage complexity was brilliant, the concept can easily be carried too far. People have a tendency to decompose complex situations into small pieces, probably a legacy from the efficiency experts who divided jobs into miniscule tasks. After decomposition, each piece is measured and each measurement is optimized. You would expect that when all of the individual measurements are optimized, then the overall system would be optimized too. But you would be wrong. If you break a value stream into silos and optimize them separately, experience has shown that the overall system will almost certainly be suboptimized.

We can't measure everything, and because we have to limit our measurements, some things invariably fall between the cracks.[30] For example, we tend to measure project performance based on cost, schedule, and scope targets. But these measurements don't take quality and customer satisfaction into account. If a conflict arises, they lose out. So what should be done? The decomposition approach would add two more measurements, so now we measure cost, schedule, scope, quality, and customer satisfaction. Voila! We have turned the iron triangle into an iron pentagon.

When a measurement system has too many measurements the real goal of the effort gets lost among too many surrogates, and there is no guidance for making tradeoffs among them. The solution is to "*Measure UP*" that is, raise the measurement one level and *decrease* the number of measurements. Find a higher-level measurement that will drive the right results for the lower level metrics *and* establish a basis for making trade-offs. In the example of project metrics above, instead of adding two more measurements we should use a single measurement—the one thing that that really matters. Return on investment

30. A great book on this topic is Rob Austin's *Measuring and Managing Performance in Organizations*, Dorset House, 1996.

Poppendieck.book Page 41 Thursday, July 27, 2006 11:46 AM

(ROI) is a possible candidate for projects. Profit and loss models work well for products.[31] If we optimize the one thing that really matters, the other numbers will take care of themselves.

Used Cars[32]

The ROI exercises in our classes are popular because they help people think about how to make tradeoff decisions that really satisfy customers. In one class, Ian and his group produced an ROI for a used vehicle remarketing company. It showed that reducing the vehicle inventory time from five days to four gave a very impressive financial return.

The story behind the ROI was even more interesting. Ian had just learned about Scrum when he was assigned to replace a very old computer system at the used vehicle remarketing company. He told his customer that his team would develop software in one-month increments, and if the customer was not happy after any increment, the engagement could be canceled.

"You told him *what*?" Ian's management was not happy. There was a hint that if Ian lost the engagement at the used vehicle remarketing company he might not have further work at the consulting firm. So Ian became very dedicated to finding a way to delight his customer every month.

By asking around, Ian found out that reducing vehicle inventory time would have positive financial impact, and in our class he got some practice in quantifying that impact. He went to his customer and got some real data, which he plugged into a simple financial model. The results helped both Ian and his customer focus development each month on the most valuable features.

After a few months, Ian wrote to tell us that his customer was so pleased with the results that his consulting firm was going to receive quite a bit more business, and his managers were delighted. Today Ian's managers have embraced agile software development, and the company considers itself one of the top agile companies in the United Kingdom.

—Mary Poppendieck

31. See Mary and Tom Poppendieck, *Lean Software Development: An Agile Toolkit*, Addison-Wesley, 2003, pp. 83–92.
32. Thanks to Ian Shimmings for letting us use this story.

Poppendieck.book Page 42 Thursday, July 27, 2006 11:46 AM

Try This

1. Which one of the seven myths seems especially poignant in your situation? Why?

   Early specification reduces waste
   The job of testing is to find defects
   Predictions create predictability
   Planning is commitment
   Haste makes waste
   There is one best way
   Optimize by decomposition

2. When should the value stream timeline start in your organization? What does it mean to "receive an order" in your environment? Some companies start the timeline for product development when a product concept is approved for development (the order comes from the approval process). Others start the timeline from the time marketing recognizes a customer need and requests a new feature. What works best for you?

3. Churn: In your organization,

   a. What percentage of requirements change after they are documented?

   b. What percentage of development time is spent in "test and fix" or "hardening" activities at the end of a development cycle?

   c. What might you do to reduce these percentages?

4. People: How much training do first-line supervisors receive about how to lead their departments? What kinds of items receive emphasis on their appraisals? How much training do first-line project managers receive about how to lead their teams? What kinds of items receive emphasis on their appraisals? Is there a difference? Why?

5. Measurements: Have everyone in your organization list on a sheet of paper the key performance measurements they believe they are evaluated against. Then compile all of the measurements into a single list, perhaps anonymously. How many measurements are on the list? Are they the right ones? Are any key measurements missing? Can you think of ways to "Measure UP?"