Introduction

This is a book of thinking tools for software development leaders. It is a toolkit for translating widely accepted lean principles into effective, agile practices that fit your unique environment. Lean thinking has a long history of generating dramatic improvements in fields as diverse as manufacturing, health care, and construction. Can it do the same for software development? One thing is clear: The field of software development has plenty of opportunity for improvement.

Jim Johnson, chairman of the Standish Group, told an attentive audience[1] the story of how Florida and Minnesota each developed its Statewide Automated Child Welfare Information System (SACWIS). In Florida, system development started in 1990 and was estimated to take 8 years and to cost $32 million. As Johnson spoke in 2002, Florida had spent $170 million and the system was estimated to be completed in 2005 at the cost of $230 million. Meanwhile, Minnesota began developing essentially the same system in 1999 and completed it in early 2000 at the cost of $1.1 million. That's a productivity difference of over 200:1. Johnson credited Minnesota's success to a standardized infrastructure, minimized requirements, and a team of eight capable people.

This is but one example of dramatic performance differences between organizations doing essentially the same thing. Such differences can be found not only in software development but in many other fields as well. Differences between companies are rooted in their organizational history and culture, their approach to the market, and their ability to capitalize on opportunities.

The difference between high-performance companies and their average competitors has been studied for a long time, and much is known about what makes some

1. Johnson, "ROI, It's Your Job."

companies more successful than others. Just as in software development, there is no magic formula, no silver bullet.[2] There are, however, some solid theories about which approaches foster high performance and which are likely to hinder it. Areas such as manufacturing, logistics, and new product development have developed a body of knowledge of how to provide the best environment for superior performance.

We observe that some methods still considered standard practice for developing software have long been abandoned by other disciplines. Meanwhile, approaches considered standard in product development, such as concurrent engineering, are not yet generally considered for software development.

Perhaps some of the reluctance to use approaches from product development comes from unfortunate uses of metaphors in the past. Software development has tried to model its practices after manufacturing and civil engineering, with decidedly mixed results. This has been due in part to a naive understanding of the true nature of these disciplines and a failure to recognize the limits of the metaphor.

While recognizing the hazards of misapplied metaphors, we believe that software development is similar to product development and that the software development industry can learn much from examining how changes in product development approaches have brought improvements to the product development process. Organizations that develop custom software will recognize that their work consists largely of development activities. Companies that develop software as a product or part of a product should find the lessons from product development particularly germane.

The story of the Florida and Minnesota SACWIS projects is reminiscent of the story of the General Motors GM-10 development, which began in 1982.[3] The first model, a Buick Regal, hit the streets seven years later, in 1989, two years late. Four years after the GM-10 program began, Honda started developing a new model Accord aimed at the same market. It was on the market by the end of 1989, about the same time the GM-10 Cutlass and Grand Prix appeared. What about quality? Our son was still driving our 1990 Accord 12 years and 175,000 mostly trouble-free miles later.

Studies[4] at the time showed that across multiple automotive companies, the product development approaches typical of Japanese automakers resulted in a 2:1 reduction in engineering effort and shortened development time by one-third when compared to traditional approaches. These results contradicted the conventional wisdom at the time, which held that the cost of change during final production was 1,000 times greater than the cost of a change made during design.[5] It was widely held

2. See Brooks, "No Silver Bullet."

3. Womack, Jones and Roos, *The Machine That Changed the World,* 110.

4. Ibid., 111.

that rapid development meant hasty decision making, so shortening the development cycle would result in many late changes, driving up development cost.

To protect against the exponentially increasing cost of change, traditional product development processes in U.S. automotive manufacturers were sequential, and relationships with suppliers were arm's length. The effect of this approach was to lengthen the development cycle significantly while making adaptation to current market trends impossible at the later stages of development. In contrast, companies such as Honda and Toyota put a premium on rapid, concurrent development and the ability to make changes late in the development cycle. Why weren't these companies paying the huge penalty for making changes later in development?

One way to avoid the large penalty for a change during final production is to make the right design decision in the first place and avoid the need to change later. That was the Detroit approach. Toyota and Honda had discovered a different way to avoid the penalty of incorrect design decisions: Don't make irreversible decisions in the first place; delay design decisions as long as possible, and when they are made, make them with the best available information to make them correctly. This thinking is very similar to the thinking behind just-in-time manufacturing, pioneered by Toyota: Don't decide what to manufacture until you have a customer order; then make it as fast as possible.

Delaying decisions is not the whole story; it is an example of how thinking differently can lead to a new paradigm for product development. There were many other differences between GM and Honda in the 1980s. GM tended to push critical decisions up to a few high-level authorities, while Honda's decision to design a new engine for the Accord emerged from detailed, engineering-level discussions over millimeters of hood slope and layout real estate. GM developed products using sequential processes, while Honda used concurrent processes, involving those making, testing, and maintaining the car in the design of the car. GM's designs were subject to modification by both marketing and strong functional managers, while Honda had a single leader who envisioned what the car should be and continually kept the vision in front of the engineers doing the work.[6]

The approach to product development exemplified by Honda and Toyota in the 1980s, typically called *lean development,* was adapted by many automobile companies in the 1990s. Today the product development performance gap among automakers has significantly narrowed.

---

5. Thomas Group, National Institute of Standards & Technology Institute for Defense Analyses.

6. Womack, Jones and Roos, *The Machine That Changed the World,* 104–110.

Lean development principles have been tried and proven in the automotive industry, which has a design environment arguably as complex as most software development environments. Moreover, the theory behind lean development borrows heavily from the theory of lean manufacturing, so lean principles in general are both understood and proven by managers in many disciplines outside of software development.

LEAN PRINCIPLES, THINKING TOOLS, AGILE PRACTICES

This book is about the application of lean principles to software development. Much is known about lean principles, and we caution that organizations have not been uniformly successful in applying them, because lean thinking requires a change in culture and organizational habits that is beyond the capability of some companies. On the other hand, companies that have understood and adopted the essence of lean thinking have realized significant, sustainable performance improvements.[7]

Principles are guiding ideas and insights about a discipline, while practices are what you actually do to carry out principles.[8] Principles are universal, but it is not always easy to see how they apply to particular environments. Practices, on the other hand, give specific guidance on what to do, but they need to be adapted to the domain. We believe that there is no such thing as a "best" practice; practices must take context into account. In fact, the problems that arise when applying metaphors from other disciplines to software development are often the result of trying to transfer the practices rather than the principles of the other discipline.

Software development is a broad discipline—it deals with Web design and with sending a satellite into orbit. Practices for one domain will not necessarily apply to other domains. Principles, however, are broadly applicable across domains as long as the guiding principles are translated into appropriate practices for each domain. This book focuses on the process of translating lean principles to agile practices tailored to individual software development domains.

At the core of this book are 22 thinking tools to aid software development leaders as they develop the agile practices that work best in their particular domain. This is not a cookbook of agile practices; it is a book for chefs who are setting out to design agile practices that will work in their domain.

There are two prerequisites for a new idea to take hold in an organization:

---

7. Chrysler, for example, adopted a lean approach to supplier management, which is credited with making significant contributions to its turnaround in the early 1990s. See Dyer, *Collaborative Advantage.*

8. Senge, *The Fifth Discipline,* 373.

- The idea must be proven to work operationally, and
- People who are considering adopting the change must understand why it works.[9]

Agile software development practices have been shown to work in some organizations, and in *Adaptive Software Development*[10] Jim Highsmith develops a theoretical basis for why these practices work. *Lean Development* further expands the theoretical foundations of agile software development by applying well-known and accepted lean principles to software development. But it goes further by providing thinking tools to help translate lean principles into agile practices that are appropriate for individual domains. It is our hope that this book will lead to wider acceptance of agile development approaches.[11]

GUIDED TOUR

This book contains seven chapters devoted to seven lean principles and thinking tools for translating each principle into agile practices. A brief introduction to the seven lean principles concludes this introduction.

1. **Eliminate waste.** Waste is anything that does not add value to a product, value as perceived by the customer. In lean thinking, the concept of waste is a high hurdle. If a component is sitting on a shelf gathering dust, that is waste. If a development cycle has collected requirements in a book gathering dust, that is waste. If a manufacturing plant makes more stuff than is immediately needed, that is waste. If developers code more features than are immediately needed, that is waste. In manufacturing, moving product around is waste. In product development, handing off development from one group to another is waste. The ideal is to find out what a customer wants, and then make or develop it and deliver exactly what they want, virtually immediately. Whatever gets in the way of rapidly satisfying a customer need is waste.
2. **Amplify learning.** Development is an exercise in discovery, while production is an

9. See Larpé and Van Wassenhove, "Learning Across Lines."

10. Highsmith, *Adaptive Software Development*.

11. Agile software development approaches include Adaptive Software Development, ASD (Highsmith, 2000); Crystal Methods (Cockburn, 2002); Dynamic Systems Development Method, DSDM (Stapleton, 2003); Feature-Driven Development, FDD (Palmer and Felsing, 2002); Scrum (Schwaber and Beedle, 2001); and Extreme Programming, XP (Beck, 2000). See Highsmith, *Agile Software Development Ecosystems* for an overview of agile approaches.

exercise in reducing variation, and for this reason, a lean approach to development results in practices that are quite different than lean production practices. Development is like creating a recipe, while production is like making the dish. Recipes are designed by experienced chefs who have developed an instinct for what works and the capability to adapt available ingredients to suit the occasion. Yet even great chefs produce several variations of a new dish as they iterate toward a recipe that will taste great and be easy to reproduce. Chefs are not expected to get a recipe perfect on the first attempt; they are expected to produce several variations on a theme as part of the learning process.[12] Software development is best conceived of as a similar learning process with the added challenge that development teams are large and the results are far more complex than a recipe. The best approach to improving a software development environment is to amplify learning.

3. **Decide as late as possible.** Development practices that provide for late decision making are effective in domains that involve uncertainty, because they provide an options-based approach. In the face of uncertainty, most economic markets develop options to provide a way for investors to avoid locking in decisions until the future is closer and easier to predict. Delaying decisions is valuable because better decisions can be made when they are based on fact, not speculation. In an evolving market, keeping design options open is more valuable than committing early. A key strategy for delaying commitments when developing a complex system is to build a capacity for change into the system.
4. **Deliver as fast as possible.** Until recently, rapid software development has not been valued; taking a careful, don't-make-any-mistakes approach has seemed to be more important. But it is time for "speed costs more" to join "quality costs more" on the list of debunked myths.[13] Rapid development has many advantages. Without speed, you cannot delay decisions. Without speed, you do not have reliable feedback. In development the discovery cycle is critical for learning: Design, implement, feedback, improve. The shorter these cycles are, the more can be learned. Speed assures that customers get what they need now, not what they needed yesterday. It also allows them to delay making up their minds about what they really want until they know more. Compressing the value stream as much as possible is a fundamental lean strategy for eliminating waste.
5. **Empower the team.** Top-notch execution lies in getting the details right, and no one understands the details better than the people who actually do the work. Involving developers in the details of technical decisions is fundamental to achiev-

12. See Ballard, "Positive vs. Negative Iteration in Design."

13. Womack, Jones and Roos, *The Machine That Changed the World,* 111.

ing excellence. The people on the front line combine the knowledge of the minute details with the power of many minds. When equipped with necessary expertise and guided by a leader, they will make better technical decisions and better process decisions than anyone can make for them. Because decisions are made late and execution is fast, it is not possible for a central authority to orchestrate activities of workers. Thus, lean practices use pull techniques to schedule work and contain local signaling mechanisms so workers can let each other know what needs to be done. In lean software development, the pull mechanism is an agreement to deliver increasingly refined versions of working software at regular intervals. Local signaling occurs through visible charts, daily meetings, frequent integration, and comprehensive testing.

6. **Build integrity in.** A system is perceived to have integrity when a user thinks, "Yes! That is exactly what I want. Somebody got inside my mind!" Market share is a rough measure of perceived integrity for products, because it measures customer perception over time.[14] Conceptual integrity means that the system's central concepts work together as a smooth, cohesive whole, and it is a critical factor in creating perceived integrity.[15] Software needs an additional level of integrity—it must maintain its usefulness over time. Software is usually expected to evolve gracefully as it adapts to the future. Software with integrity has a coherent architecture, scores high on usability and fitness for purpose, and is maintainable, adaptable, and extensible. Research has shown that integrity comes from wise leadership, relevant expertise, effective communication, and healthy discipline; processes, procedures, and measurements are not adequate substitutes.
7. **See the whole.** Integrity in complex systems requires a deep expertise in many diverse areas. One of the most intractable problems with product development is that experts in any area (e.g., database or GUI) have a tendency to maximize the performance of the part of the product representing their own specialty rather than focusing on overall system performance. Quite often, the common good suffers if people attend first to their own specialized interests. When individuals or organizations are measured on their specialized contribution rather than overall performance, suboptimization is likely to result. This problem is even more pronounced when two organizations contract with each other, because people will naturally want to maximize the performance of their own company. It is challenging to implement practices that avoid suboptimization in a large organization, and it is an order of magnitude more difficult when contracts are involved.

14. Clark and Fujimoto, "The Power of Product Integrity," 278.
15. Brooks, *Mythical Man Month,* 255.

This book was written for software development managers, project managers, and technical leaders. It is organized around the seven principles of lean thinking. Each chapter discusses the lean principle and then provides thinking tools to assist in translating the lean principle to agile software development practices that match the needs of individual domains. At the end of each chapter are practical suggestions for implementing the lean principle in a software development organization. The last chapter is an instruction and warranty card for using the thinking tools in this toolkit.