# Lean Software Development

## An Agile Toolkit

The Agile Software Development Series

Cockburn • Highsmith
Series Editors

- Adapting agile practices to your development organization
- Uncovering and eradicating waste throughout the software development lifecycle
- Practical techniques for every development manager, project manager, and technical leader

*Forewords by*
***Jim Highsmith***
***and Ken Schwaber***

**Mary Poppendieck**
**Tom Poppendieck**

# *Lean Software Development*

# The Agile Software Development Series

Alistair Cockburn and Jim Highsmith, Series Editors


Addison-Wesley

Visit **informit.com/agileseries** for a complete list of available publications.

---

Agile software development centers on four values, which are identified in the Agile Alliance's Manifesto:

1. Individuals and interactions over processes and tools
2. Working software over comprehensive documentation
3. Customer collaboration over contract negotiation
4. Responding to change over following a plan

The development of Agile software requires innovation and responsiveness, based on generating and sharing knowledge within a development team and with the customer. Agile software developers draw on the strengths of customers, users, and developers to find just enough process to balance quality and agility.

The books in The Agile Software Development Series focus on sharing the experiences of such Agile developers. Individual books address individual techniques (such as Use Cases), group techniques (such as collaborative decision making), and proven solutions to different problems from a variety of organizational cultures. The result is a core of Agile best practices that will enrich your experiences and improve your work.

# *Lean Software Development*

## *An Agile Toolkit*

***Mary Poppendieck***
***Tom Poppendieck***

Addison
Wesley

Boston • San Francisco • New York • Toronto • Montreal
London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Pearson Education Corporate Sales Division
One Lake Street
Upper Saddle River, NJ 07458
(800) 382-3419
corpsales@pearsontechgroup.com

Visit AW on the Web: www.awl.com/cseng/

*Library of Congress Cataloging-in-Publication Data available*



ISBN 0-321-15078-3
Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
Seventeenth printing, May 2013

*To Dustin, Andy and Brian, Karen and Becca*

*This page intentionally left blank*

# *Contents*

# *Foreword*

**BY JIM HIGHSMITH**

In February 2001, when "Agile" was adopted as the umbrella word for methodologies such as Extreme Programming, Crystal, Adaptive Software Development, Scrum, and others, the industrial heritage of agile buzzed around in the background. Womack, Jones, and Roos's *The Machine That Changed the World*, Smith and Reinertsen's *Developing Products in Half the Time*, and Womack and Jones's *Lean Thinking* have resided on my bookshelf for years. The Agility Forum was founded by manufacturers in the early 1990s. The extensive literature on agile and lean industrial product development influenced my work on *Adaptive Software Development*.

But in *Lean Software Development,* Mary and Tom Poppendieck take lean industrial practices to a new level—they tell us how to apply them directly to software development. It is one thing to read about value stream mapping in a manufacturing plant but quite another to see how this idea applies to software development processes. It is one thing to read about Toyota's set-based decision making and another to apply those ideas to software design. Mary's manufacturing and industrial product development experience at 3M gives her insight into how these practices actually work, and her and Tom's information technology backgrounds gives them insight into how to apply the practices to software development.

Although Agile Software Development has roots that go back more than 10 years, as a movement it is only a couple of years old (in early 2003). Tying it to lean and agile industrial product development provides additional credibility to the principles and practices of Agile Software Development, but more importantly, it provides a wealth of ideas that can strengthen agile practices.

For example, the set-based decision making previously mentioned counters prevalent ideas about making design decisions. Traditional engineering (software and others) stresses analysis and early decision making so downstream activities can proceed. Set-based development stresses keeping multiple design options open in order to have as much information as possible, not only about a particular piece of the design, but also about the integration of all pieces. Set-based development helps optimize the whole rather than the pieces. Simple design and refactoring serve similar purposes for software developers—pushing off certain design decisions into the future when more information is available. Set-based development therefore provides a parallel that adds credibility to agile practices but also shows how to extend those practices.

*Lean Software Development* provides a wealth of information about applying lean techniques from an industrial setting to software development. In particular, it presents a toolkit for project managers, team leaders, and technology managers who want to add value rather than become roadblocks to their project teams.

Jim Highsmith
Flagstaff, Arizona
March 2002

# *Foreword*

***BY KEN SCHWABER***

Agile processes for software development came into being during the 1990's. We constructed them based on experience, trial-and-error, knowledge of what didn't work, and best practices. I had used Scrum and Extreme Programming-like practices in my own software company during the early 90's. When I first formulated the detailed practices of Scrum, I made sure that I tried them on every sort of development situation imaginable before I published my first book about Scrum. In the absence of first-principles or a theoretical framework for Scrum and other agile processes, I wanted to make sure it really worked before I unleashed more snake oil on the world.

Others and I have made attempts to provide a theoretical underpinning to agile processes. I've referred back to my research in industrial process control theory, which friends of mine at DuPont's Advanced Research Facility helped me understand and apply. Jim Highsmith has referred to the principles of complex adaptive systems and complexity theory to explain, by analogy, the reasons why agile processes work.

Mary and Tom Poppendieck have provided us with a more understandable, robust, and everyday framework for understanding the workings of agile processes. I was with them at the XP2002 conference in Sardinia, Italy when Enrico Zaninotto, Dean of Faculty of Economics at the University of Trento, Italy gave his keynote talk, "From X Programming to the X Organization." In this talk, Enrico laid out the migration of manufacturing from the simple workshop through the assembly line to the modern use of lean manufacturing. He clearly demonstrated the economic imperatives underlying the current use of lean manufacturing. After the talk, Mary was obviously pleased at this validation. Enrico's talk brought together her background in manufacturing and product de-

velopment with all of the collaborative work she had done with the lean construction movement and her knowledge of the Toyota production system.

This book is the consequence of the Poppendiecks' work to pull all of these movements and knowledge together. In doing so, they have provided a commonsense set of tools that underlie agile processes. People using agile processes can refer to the 22 tools that Mary and Tom describe to understand why and how the most common agile processes work, or to modify them based on a deep understanding of them, or to construct their own agile process. The tools in this book provide the framework.

I took particular pleasure in listening to Enrico and seeing Mary's and Tom's thinking gel. Our industry has long been burdened by the accusation that we should be able to "do it like manufacturing!" The manufacturing this referred to was the Frederick Taylor, Henry Ford assembly line. The systems development processes we constructed on Taylor's principles didn't work, and we didn't know why. Enrico laughed—"Modern manufacturing left the Taylor principles behind twenty years ago!"

No longer do we need to refer to such abstruse theory and science as complex adaptive systems to explain agile systems development. We can refer to the 22 tools set forth in this book and look to manufacturing and common sense for their rationale. We are finally starting to model software development on something that works for us!

Ken Schwaber
February 2003

# *Preface*

I used to be a really good programmer. My code controlled telephone switching systems, high energy physics research, concept vehicles, and the makers and coaters used to manufacture 3M tape. I was equally good at writing Fortran or assembly language, and I could specify and build a minicomputer control system as fast as anyone.

After a dozen or so years of programming, I followed one of my systems to a manufacturing plant and took the leap into IT management. I learned about materials control and unit costs and production databases. Then the quality-is-free and just-in-time movements hit our plant, and I learned how a few simple ideas and empowered people could change everything.

A few years later I landed in new product development, leading commercialization teams for embedded software, imaging systems, and eventually optical systems. I liked new product development so much that I joined a start-up company and later started my own company to work with product development teams, particularly those doing software development.

I had been out of the software development industry for a half dozen years, and I was appalled at what I found when I returned. Between PMI (Project Management Institute) and CMM (Capability Maturity Model) certification programs, a heavy emphasis on process definition and detailed, front-end planning seemed to dominate everyone's perception of best practices. Worse, the justification for these approaches was the lean manufacturing movement I knew so well.

I was keenly aware that the success of lean manufacturing rested on a deep understanding of what creates value, why rapid flow is essential, and how to release the brainpower of the people doing the work. In the prevailing focus on process and planning I detected a devaluation of these key principles. I heard, for example, that detailed process definitions were needed so that "anyone can program," while lean

manufacturing focused on building skill in frontline people and having them define their own processes.

I heard that spending a lot of time and getting the requirements right upfront was the way to do things "right the first time." I found this curious. I knew that the only way that my code would work the first time I tried to control a machine was to build a complete simulation program and test the code to death. I knew that every product that was delivered to our plant came with a complete set of tests, and "right the first time" meant passing each test every step of the way. You could be sure that next month a new gizmo or tape length would be needed by marketing, so the idea of freezing a product configuration before manufacturing was simply unheard of. That's why we had serial numbers—so we could tell what the current manufacturing spec was the day a product was made. We would never expect to be making the exact same products this month that we were making last month.

Detailed front-end planning strikes me as diametrically opposed to lean manufacturing principles. Process definition by a staff group strikes me as diametrically opposed to the empowerment that is core to successful lean manufacturing. It seems to me that the manufacturing metaphor has been misapplied to software development. It seems to me that CMM, in its eagerness to standardize process, leaves out the heart of discovery and innovation that was the critical success factor in our move to total quality management. We knew in manufacturing that ISO9000 and even Malcolm Baldrige awards had little or nothing to do with a successful quality program. They were useful in documenting success, but generally got in the way of creating it.

It seems to me that a PMI certification program teaches a new project manager several antipatterns for software project management. Work breakdown. Scope control. Change control. Earned value. Requirements tracking. Time tracking. I learned all about these when I was a program manager for government contracts at 3M, and was keenly aware of the waste they added to a program. We certainly knew better than to use them on our internal product development programs, where learning and innovation were the essential ingredients of success.

This is not to say that CMM and PMI are bad, but only that for anyone who has lived through the lean revolution, they tend to give the wrong flavor to a software development program. In this book we hope to change the software development paradigm from process to people, from disaggregation to aggregation, from speculation to data-based decision making, from planning to learning, from traceability to testing, from cost-and-schedule control to delivering business value.

If you think that better, cheaper, and faster can't coexist, you should know that we used to think the same way in the pre-lean days of manufacturing and product development. However, we learned that by focusing on value, flow, and people, you got

better quality, lower cost, and faster delivery. We learned that from our competitors as they took away our markets.

May you lead your industry in lean software development.

Mary Poppendieck

## ACKNOWLEDGMENTS

This book is in our words, but the ideas came from the agile community. Lean principles have had decades of success in lean manufacturing, logistics, and construction. These same principles, which are the framework of this book, are finally emerging as agile software development.

Many reviewers invested thoughtful hours reading and providing feedback that helped us refine our ideas and presentation, including Ken Schwaber, Jim Highsmith, Alistair Cockburn, Luke Hohmann, Martin Fowler, pragmatic Dave Thomas, Bill Wake, Rob Purser, Mike Cohn, and Martha Lindeman. Thanks to Glenn Ballard and Greg Howell from the Lean Construction Institute for their contributions. We also thank Kent Beck, Tim Ocock, Ron Crocker, and Bruce Ferguson for their contributions to the book.

Thanks to our employers, mentors, team members, collaborators and clients. Thanks to all who have attended our classes and tutorials, asked probing questions, and given us examples of lean principles that work (or do not work) in their worlds. Finally, thanks to the authors of the books and articles we cited, for their contributions to agile software development.

*This page intentionally left blank*

# *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 adopted 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, *The 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.

# Chapter 3

# *Decide as Late as Possible*

## CONCURRENT DEVELOPMENT[1]

When sheet metal is formed into a car body, a massive stamping machine presses the metal into shape. The stamping machine has a huge metal die, which makes contact with the sheet metal and presses it into the shape of a fender, door, or another body panel. Designing and cutting the dies to the proper shape accounts for half of the capital investment of a new car development program and drives the critical path. If a mistake ruins a die, the entire development program suffers a huge setback. If there is one thing that automakers want to do right, it is the die design and cutting.

The problem is, as the car development progresses, engineers keep making changes to the car, and these find their way to the die design. No matter how hard the engineers try to freeze the design, they are not able to do so. In Detroit in the 1980s the cost of changes to the design was 30 to 50 percent of the total die cost, while in Japan it was 10 to 20 percent. These numbers seem to indicate the Japanese companies must have been much better at preventing change after the die specs were released to the tool and die shop. But such was not the case.

The U.S. strategy for making a die was to wait until the design specs were frozen, and then send the final design to the tool and die maker, which triggered the process of ordering the block of steel and cutting it. Any changes went through an arduous change approval process. It took about two years from ordering the steel to the time the die would be used in production. In Japan, however, the tool and die makers order

1. Information drawn from Womack, Jones and Roos, *The Machine That Changed the World*, 116–119, and Clark and Fujimoto, *Product Development Performance*, 187, 236–237.

up the steel blocks and start rough cutting at the same time the car design is starting. This is called concurrent development. How can it possibly work?

The die engineers in Japan are expected to know a lot about what a die for a front door panel will involve, and they are in constant communication with the body engineer.[2] They anticipate the final solution, and they are also skilled in techniques to make minor changes late in development, such as leaving more material where changes are likely. Most of the time die engineers are able to accommodate the engineering design as it evolves. In the rare case of a mistake, a new die can be cut much faster because the whole process is streamlined.

Japanese automakers do not freeze design points until late in the development process, allowing most changes to occur while the window for change is still open. When compared to the early design freeze practices in the United States in the 1980s, Japanese die makers spent perhaps a third as much money on changes and produced better die designs. Japanese dies tended to require fewer stamping cycles per part, creating significant production savings.[3]

The impressive difference in time-to-market and increasing market success of Japanese automakers prompted U.S. automotive companies to adopt concurrent development practices in the 1990s, and today the product development performance gap has narrowed significantly.

## Concurrent Software Development

Programming is a lot like die cutting. The stakes are often high, and mistakes can be costly, so sequential development, that is, establishing requirements before development begins, is commonly thought of as a way to protect against serious errors. The problem with sequential development is that it forces designers to take a depth-first rather than a breadth-first approach to design. Depth-first forces making low-level dependent decisions before experiencing the consequences of the high-level decisions. The most costly mistakes are made by forgetting to consider something important at the beginning. The easiest way to make such a big mistake is to drill down to detail too fast. Once you set down the detailed path, you can't back up and are un-

2. The close collaboration between Japanese die engineer and designer occurs even though the die engineer is an external supplier. Changes are anticipated in the contract and are done on a worker-to-worker basis without the delay of a change approval process. We discuss contracts that allow for such close collaboration in Chapter 7, "See the Whole."

3. Typical Japanese stamping in 1990 took five shots per panel, compared to seven in the United States (Clark and Fujimoto, *Product Development Performance*, 186).

likely to realize that you should. When big mistakes may be made, it is best to survey the landscape and delay the detailed decisions.

Concurrent development of software usually takes the form of iterative development. It is the preferred approach when the stakes are high and the understanding of the problem is evolving. Concurrent development allows you to take a breadth-first approach and discover those big, costly problems before it's too late. Moving from sequential development to concurrent development means you start programming the highest value features as soon as a high-level conceptual design is determined, even while detailed requirements are being investigated. This may sound counterintuitive, but think of it as an exploratory approach that permits you to learn by trying a variety of options before you lock in on a direction that constrains implementation of less important features.

In addition to providing insurance against costly mistakes, concurrent development is the best way to deal with changing requirements, because not only are the big decisions deferred while you consider all the options, but the little decisions are deferred as well. When change is inevitable, concurrent development reduces delivery time and overall cost while improving the performance of the final product.

If this sounds like magic—or hacking—it would be if nothing else changed. Just starting programming earlier, without the associated expertise and collaboration found in Japanese die cutting, is unlikely to lead to improved results. There are some critical skills that must be in place in order for concurrent development to work.

Under sequential development, U.S. automakers considered die engineers to be quite remote from the automotive engineers, and so too, programmers in a sequential development process often have little contact with the customers and users who have requirements and the analysts who collect requirements. Concurrent development in die cutting required U.S. automakers to make two critical changes—the die engineer needed the expertise to anticipate what the emerging design would need in the cut steel and had to collaborate closely with the body engineer.

Similarly, concurrent software development requires developers with enough expertise in the domain to anticipate where the emerging design is likely to lead and close collaboration with the customers and analysts who are designing how the system will solve the business problem at hand.

## Cost Escalation

Software is different from most products in that software systems are expected to be upgraded on a regular basis. On the average, more than half of the development work that occurs in a software system occurs after it is first sold or placed into production.[4] In addition to internal changes, software systems are subject to a changing environment—a new operating system, a change in the underlying database, a change in the

client used by the GUI, a new application using the same database, and so on. Most software is expected to change regularly over its lifetime, and in fact once upgrades are stopped, software is often nearing the end of its useful life. This presents us with a new category of waste: waste caused by software that is difficult to change.

In 1987 Barry Boehm wrote, "Finding and fixing a software problem after delivery costs 100 times more than finding and fixing the problem in early design phases."[5] This observation became the rationale behind thorough upfront requirements analysis and design, even though Boehm himself encouraged incremental development over "single-shot, full product development."[6] In 2001 Boehm noted that for small systems the escalation factor can be more like 5:1 than 100:1; and even on large systems, good architectural practices can significantly reduce the cost of change by confining features that are likely to change to small, well-encapsulated areas.[7]

There used to be a similar, but more dramatic, cost escalation factor for product development. It was once estimated that a change after production began could cost 1,000 times more than if the change had been made in the original design.[8] The belief that the cost of change escalates as development proceeds contributed greatly to standardizing the sequential development process in the United States. No one seemed to recognize that the sequential process could actually be the ***cause*** of the high escalation ratio. However, as concurrent development replaced sequential development in the United States in the 1990s, the cost escalation discussion was forever altered. It was no longer how much a change might cost later in development; the discussion centered on how to reduce the need for change through concurrent engineering.

Not all change is equal. There are a few basic architectural decisions that you need to get right at the beginning of development, because they fix the constraints of the system for its life. Examples of these may be choice of language, architectural layering decisions, or the choice to interact with an existing database also used by other applications. These kinds of decisions might have the 100:1 cost escalation ratio. Because these decisions are so crucial, you should focus on minimizing the number of

4. The percentage of software lifecycle cost attributed to maintenance ranges between 40 and 90 percent. See Kajko-Mattsson et al., "Taxonomy of Problem Management Activities."

5. Boehm, "Industrial Software Metrics Top 10 List."

6. Boehm and Papaccio, "Understanding and Controlling Software Costs," 1465–1466.

7. Boehm and Basili, "Software Defect Reduction List."

8. Concurrent engineering has been credited with reducing product development time by 30 to 70 percent, engineering changes by 65 to 90 percent, and time to market by 20 to 90 percent, while improving quality by 200 to 600 percent and productivity by 20 to 110 percent (Thomas Group, 1990).

these high-stakes constraints. You also want to take a breadth-first approach to these high-stakes decisions.

The bulk of the change in a system does not have to have a high-cost escalation factor; it is the sequential approach that causes the cost of most changes to escalate exponentially as you move through development. Sequential development emphasizes getting all the decisions made as early as possible, so the cost of all changes is the same—very high. Concurrent design defers decisions as late as possible. This has four effects:

- Reduces the number of high-stake constraints.
- Gives a breadth-first approach to high-stakes decisions, making it more likely that they will be made correctly.
- Defers the bulk of the decisions, significantly reducing the need for change.
- Dramatically decreases the cost escalation factor for most changes.

A single cost escalation factor or curve is misleading.[9] Instead of a chart showing a single trend for all changes, a more appropriate graph has at least two cost escalation curves, as shown in Figure 3.1. The agile development objective is to move as many changes as possible from the top curve to the bottom curve.

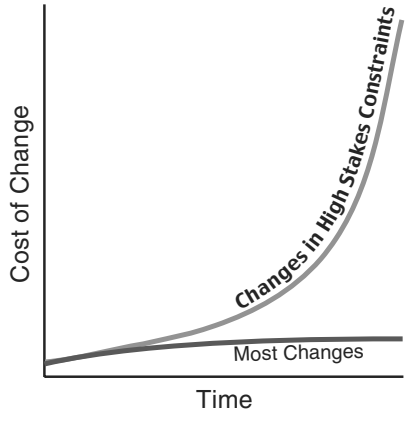


***Figure 3.1*** *Two cost escalation curves.*

9. The cost escalation number of 100:1 in Boehm and Papaccio, "Understanding and Controlling Software Costs," refers to the cost of fixing or reworking software. In Beck, *Extreme Programming Explained*, the cost escalation curve on page 23 refers to all change, not just fixing or rework.

Returning for a moment to the die cutting example, the die engineer sees the conceptual design of the car and knows roughly the necessary door panel size. With that information, a big enough steel block can be ordered. If the concept of the car changes from a small, sporty car to a mid-size family car, the block of steel may be too small, and that would be a costly mistake. But the die engineer knows that once the overall concept is approved, it won't change, so the steel can be safely ordered long before the details of the door emerge. Concurrent design is a robust design process because the die adapts to whatever design emerges.

Lean software development delays freezing all design decisions as long as possible, because it is easier to change a decision that hasn't been made. Lean software development emphasizes developing a robust, change-tolerant design, one that accepts the inevitability of change and structures the system so that it can be readily adapted to the most likely kinds of changes.

The main reason software changes throughout its lifecycle is that the business process in which it is used evolves over time. Some domains evolve faster than others, and some domains may be essentially stable. It is not possible to build in flexibility to accommodate arbitrary changes cheaply. The idea is to build tolerance for change into the system along domain dimensions that are likely to change. Observing where the changes occur during iterative development gives a good indication of where the system is likely to need flexibility in the future.[10] If changes of certain types are frequent during development, you can expect that these types of changes will not end when the product is released. The secret is to know enough about the domain to maintain flexibility, yet avoid making things any more complex than they must be.

If a system is developed by allowing the design to emerge through iterations, the design will be robust, adapting more readily to the types of changes that occur during development. More importantly, the ability to adapt will be built in to the system so that as more changes occur after its release, they can be readily incorporated. On the other hand, if systems are built with a focus on getting everything right at the beginning in order to reduce the cost of later changes, their design is likely to be brittle and not accept changes readily. Worse, the chance of making a major mistake in the key structural decisions is increased with a depth-first rather than a breadth-first approach.

## TOOL 7: OPTIONS THINKING

"Satisfaction Guaranteed or Your Money Back." Sears. Target. L.L. Bean. Land's End. Amazon.com. What store doesn't guarantee satisfaction? On the other hand, it's

10. See the discussion on developing a sense of how to absorb changes in "Tool 8: The Last Responsible Moment" later in this chapter.

somewhat scary to think about offering a satisfaction guarantee for software. Usually, the message is, After you open the shrink wrap—or after you sign off the requirements—it's yours. Software rarely comes with a warranty.

Let's step to the other side of this transaction and consider why *satisfaction guaranteed* warranties are so attractive. The underlying dynamic is that people find it difficult to make irrevocable decisions when there is uncertainty present. For example, if you are buying a gift and aren't certain about the recipient's size or color preference, a satisfaction guarantee lets you purchase before you get the answers. You are not being asked to make an irrevocable decision until the uncertainty is resolved. You have the right to return the product, usually in new condition in a set timeframe, so if it doesn't work out, you don't lose any more than the time and effort required to evaluate and return it.

It would be nice if business transactions came with a *satisfaction guaranteed* clause, but they rarely do. Most business decisions are irrevocable; we usually don't have the option to change our mind. Interestingly enough, even though we would like to be able to change our mind, we usually don't give our customers the option to change their minds. And yet, almost everyone resists making irrevocable decisions in the face of uncertainty. It would be nice if we could find a way to delay making decisions and a way to provide the same benefit for our customers.

## Delaying Decisions

Hewlett-Packard discovered a way to increase profits by delaying decisions. HP sells a lot of printers around the world, and in many countries, the electrical connection must be tailored to the local electrical outlets. You would think that HP could accurately forecast how many printers it would sell in each country, but the forecasts are always just a bit off. HP always had some excess printers for one country and not enough for another. Then the company hit upon the idea of doing final electrical configuration in the warehouse after the printer was ordered. It costs more to configure a printer in a warehouse than in the factory, but overall, the cost of the option to customize was more than offset by the benefit of always having the right product. Even though unit costs rose, HP saved $3 million a month by more effectively matching supply to demand.[11]

As a keynote speaker at a software conference,[12] Enrico Zaninotto, an Italian economist, pointed out that the underlying economic mechanism for controlling complexity in just-in-time systems is minimizing irreversible actions. What does this

11. Coy, "Exploring Uncertainty."

12. Zaninotto, "From X Programming to the X Organization."

mean? In HP's case, there was a huge amount of complexity involved in getting the right electrical connection on printers going to different countries. The approach used to control this complexity was delaying the decision about what electrical connection to install until after an order was received in the warehouse. Voila! The system was no longer so complex.

Zaninotto contrasted just-in-time systems with Fordist mass production systems, which manage complexity by limiting the number of options—"You can have any color as long as it's black." For example, a mass production system for printers would use only the most common type of plug and make differences in plug styles the customers' problem; they can buy an add-on converter at their local electronic shop.

Zaninotto suggested that when a system that prespecifies options is confronted by a system that keeps options open, the second system wins out in a complex dynamic market. Thus, once HP started customizing power options on printers, the market expected all the other manufacturers to do the same.

Delaying irreversible decisions until uncertainty is reduced has economic value. It leads to better decisions, it limits risk, it helps manage complexity, it reduces waste, and it makes customers happy. On the other hand, delaying decisions usually comes at a cost. In HP's case, the unit cost of adding a cord in the warehouse was higher than the cost of adding the cord in the factory. Still, the overall system was more profitable, because delaying decisions allowed the correct decision to be made every time.

## Options

The financial and commodities markets have developed a mechanism—called *options*—to allow decisions to be delayed. An option is the right, but not the obligation, to do something in the future. It's like a *satisfaction guaranteed* warranty—if things work out the way you expect them to, you can exercise the option (equivalent to keeping the product). If things don't work out, you can ignore the option (equivalent to returning the product), and all you lose is whatever the option cost you in the first place.

Uncertainty can move in two directions—unexpected good things can happen just as easily as unexpected bad things. No one knows this better than farmers, who have to deal with rising and falling commodity prices. Starting in 1985, the Chicago Board of Trade started selling options contracts, which provide farmers with commodity price insurance. Farmers can now buy options that guarantee a minimum price for their crop and still be free to sell it at a higher price if the market goes up.

A hotel reservation is an option on a hotel room in the future. The price of the option is the cost to make the reservation, which may include a reservation fee. If you exercise the option—if you show up at the hotel—you pay the price negotiated at the time the reservation was made. If you cancel the trip all you lose is the reservation fee.

Stock options are a way to give employees an opportunity to profit if the company does well in the future while limiting their risk if the company does poorly.[13] In general, financial options give the buyer an opportunity to capitalize on positive events in the future while limiting exposure to negative events. Options provide opportunities to make decisions down the road, while providing insurance against things going wrong.

## *Microsoft Strategy, circa 1988*

In 1999 Eric Beinhocker[14] reminisced about the 1988 Comdex trade show. All the big players were there with big booths: Apple was at the peak of its powers; IBM, Hewlett-Packard, DEC, Apollo, and Sun Microsystems were all touting their latest strategies. And then there was Microsoft, with a modest booth that "was more like a Middle Eastern bazaar than a trade-show booth." Microsoft showed its then current strength, DOS, along with an early version of Windows, OS/2 for IBM machines, a version of UNIX, and new releases of Word and Excel, which were a far distant second to Lotus and WordPerfect in the DOS environment but led the applications on Apple platforms. Beinhocker notes: "Along with confused customers, the press was also grumbling. Columnists claimed that Microsoft was adrift and Gates had no strategy."

In 1988 it was not at all clear which platform would win, and Gates did have a strategy—to cover all the bases. He wanted Windows to win but hedged his bets with DOS, OS/2, and even a version of UNIX. If Apple won the war, he would lose the operating system but win as the dominant application provider on that platform. In any case, he would develop expertise in both operating systems and applications. He played the options game and let the market emerge.

Microsoft was not the only company to invest in options. IBM's strategy was to offer multiple options in hardware, thus the introduction of the PC in 1981. IBM did not fully realize that it was the software business, not the hardware business, that would become the economic driver of the future, so it had allowed Microsoft to hold most of the options in the software market. This made sense at the time, since all options come at a price, but IBM chose the wrong options.

In the 1990s Cisco Systems acquired companies with relevant technologies rather than maintaining a large research and development effort. This allowed Cisco to delay selecting technologies until both the market and the technology emerged, considerably reducing its risk. The cost of this options-based approach was the premium paid for the companies that had born the initial risk.

---

13. This assumes that the options are in addition to a reasonable salary.

14. Beinhocker, "Robust Adaptive Strategies," 95–96.

## Options Thinking in Software Development

One of the hot debates in software development concerns the tradeoff between predictive processes and adaptive processes. The prevailing paradigm has been a predictive process: Software development should be specified in detail prior to implementation, because if you don't get the requirements nailed down and the design right, it will surely cost a lot to make changes later. This paradigm may work in a highly predictable world. However, if there is uncertainty about what customers really need, whether their situation will change, or where technology is moving, then an adaptive approach is a better bet. Options limit downside risk by limiting the cost and time allocated to resolving uncertainty. They maximize upside reward by delaying decisions until more knowledge is available. Economists and manufacturing managers alike understand that the adaptive paradigm of delaying decisions until uncertainty is reduced usually produces better results than a predictive approach.

Agile software development processes can be thought of as creating options that allow decisions to be delayed until the customer needs are more clearly understood and evolving technologies have had time to mature. This is not to say that agile approaches are unplanned. Plans help clarify confusing situations, allow consideration of tradeoffs, and establish patterns that allow rapid action. So, plans tend to enhance the flexibility to respond to change. However, *a plan should not prespecify detailed actions based on speculation.* Agile software development follows speculation with experiments and learning to reduce uncertainty and adapt the plan to reality.[15]

Conventional wisdom in software development tends to generate detailed decisions early in the process—like freezing the customer requirements and specifying the technical framework. In this approach, what is taken for planning is usually a process of predicting the future and making early decisions based on those predictions without any data or validation. Plans and predictions are not bad, but making irrevocable decisions based on speculation is to be avoided.

In 1988 Harold Thimbleby published a paper in *IEEE Software* titled "Delaying Commitment." He notes that when faced with a new situation, experts will delay firm decisions while they investigate the situation, because they know that delaying commitments often leads to new insights. Amateurs, on the other hand, want to get everything completely right, so they tend to make early decisions, quite often the wrong ones. Once these early decisions are made, other decisions are built on them, making them devilishly difficult to change. Thimbleby notes that premature design commitment is a design failure mode that restricts learning, exacerbates the impact of defects, limits the usefulness of the product, and increases the cost of change.

---

15. See Highsmith, *Adaptive Software Development*, 41–48.

Options thinking is an important tool in software development as long as it is accompanied by recognition that options are not free and it takes expertise to know which options to keep open. Options do not guarantee success; they set the stage for success if the uncertain future moves in a favorable direction. Options allow fact-based decisions based on learning rather than speculation.

## TOOL 8: THE LAST RESPONSIBLE MOMENT

Concurrent software development means starting developing when only partial requirements are known and developing in short iterations that provide the feedback that causes the system to emerge. Concurrent development makes it possible to delay commitment until the *last responsible moment*,[16] that is, the moment at which failing to make a decision eliminates an important alternative. If commitments are delayed beyond the last responsible moment, then decisions are made by default, which is generally not a good approach to making decisions.

Procrastinating is not the same as making decisions at the last responsible moment; in fact, delaying decisions is hard work. Here are some tactics for making decisions at the last responsible moment:

***Share partially complete design information.*** The notion that a design must be complete before it is released is the biggest enemy of concurrent development. Requiring complete information before releasing a design increases the length of the feedback loop in the design process and causes irreversible decisions to be made far sooner than necessary. Good design is a discovery process, done through short, repeated exploratory cycles.

***Organize for direct, worker-to-worker collaboration.*** Early release of incomplete information means that the design will be refined as development proceeds. This requires that people who understand the details of what the system must do to provide value must communicate directly with people who understand the details of how the code works.

***Develop a sense of how to absorb changes.*** In "Delaying Commitment" Harold Thimbleby observes that the difference between amateurs and experts is that experts know how to delay commitments and how to conceal their errors for as long as possible. Experts repair their errors before they cause problems. Amateurs try to get everything right the first time and so overload their problem-

16. The Lean Construction Institute coined the term *last responsible moment*. See *www.leanconstruction.org*.

solving capacity that they end up committing early to wrong decisions. Thimbleby recommends some tactics for delaying commitment in software development, which could be summarized as an endorsement of object-oriented design and component-based development:

- **Use modules:** Information hiding, or more generally behavior hiding, is the foundation of object-oriented approaches. Delay commitment to the internal design of the module until the requirements of the clients on the interfaces stabilize.
- **Use interfaces:** Separate interfaces from implementations. Clients should not depend on implementation decisions.
- **Use parameters:** Make magic numbers—constants that have meaning—into parameters. Make magic capabilities like databases and third-party middleware into parameters. By passing capabilities into modules wrapped in simple interfaces, your dependence on specific implementations is eliminated and testing becomes much easier.
- **Use abstractions:** Abstraction and commitment are inverse processes. Defer commitment to specific representations as long as the abstract will serve immediate design needs.
- **Avoid sequential programming:** Use declarative programming rather than procedural programming, trading off performance for flexibility. Define algorithms in a way that does not depend on a particular order of execution.
- **Beware of custom tool building:** Investment in frameworks and other tooling frequently requires committing too early to implementation details that end up adding needless complexity and seldom pay back. Frameworks should be extracted from a collection of successful implementations, not built on speculation.

Additional tactics for delaying commitment include

- **Avoid repetition:** This is variously known as the Don't Repeat Yourself (DRY)[17] or Once And Only Once (OAOO)[18] principle. If every capability is expressed in only one place in the code, there will be only one place to change when that capability needs to evolve, and there will be no inconsistencies.
- **Separate concerns:** Each module should have a single, well-defined responsibility. This means that a class will have only one reason to change.[19]

17. Hunt and Thomas, *The Pragmatic Programmer*, 27.

18. Beck, *Extreme Programming Explained*, 109.

19. Martin, *Agile Software Development Principles, Patterns, and Practices*, Chapter 8, calls this the Single Responsibility Principle.

- **Encapsulate variation:** What is likely to change should be inside; the interfaces should be stable. Changes should not cascade to other modules. This strategy, of course, depends on a deep understanding of the domain to know which aspects will be stable and which variable. By application of appropriate patterns, it should be possible to extend the encapsulated behavior without modifying the code itself.[20]
- **Defer implementation of future capabilities:** Implement only the simplest code that will satisfy immediate needs rather than putting in capabilities you "know" you will need in the future.[21] You will know better in the future what you really need then, and simple code will be easier to extend if necessary.
- **Avoid extra features:** If you defer adding features you "know" you will need, then you certainly want to avoid adding extra features "just-in-case" they are needed. Extra features add an extra burden of code to be tested, maintained, and understood. Extra features add complexity, not flexibility.

Much has been written on these delaying tactics,[22] so they are not covered in detail in this book.

***Develop a sense of what is critically important in the domain.*** Forgetting some critical feature of the system until too late is the fear that drives sequential development. If security, or response time, or failsafe operation are critically important in the domain, these issues need to be considered from the start; if they are ignored until too late, it will indeed be costly. However, the assumption that sequential development is the best way to discover these critical features is flawed. In practice, early commitments are more likely to overlook such critical elements than late commitments, because early commitments rapidly narrow the field of view.

***Develop a sense of when decisions must be made.*** You do not want to make decisions by default, or you have not delayed them. Certain architectural concepts such as usability design, layering, and component packaging are best made early so as to facilitate emergence in the rest of the design. A bias toward late commitment must not degenerate into a bias toward no commitment. You need to develop a keen sense of timing and a mechanism to cause decisions to be made when their time has come.

---

20. Ibid. Chapter 9 describes how to do this in the Open Closed Principle implemented via the Strategy or Template pattern.

21. Beck, *Extreme Programming Explained*, Chapter 17, uses the acronym YAGNI (You Aren't Going to Need It) for this practice and explains its rationale.

22. See Fowler, Patterns of *Enterprise Application Architecture*; Larman, *Applying UML and Patterns*; as well as the works cited above.

***Develop a quick response capability.*** The slower you respond, the earlier you have to make decisions. Dell, for instance, can assemble computers in less than a week, so it can decide what to make less than a week before shipping. Most other computer manufacturers take a lot longer to assemble computers, so they have to decide what to make much sooner. If you can change your software quickly, you can wait to make a change until customers know what they want.

## TOOL 9: MAKING DECISIONS

### Depth-First Versus Breadth-First Problem Solving

There are two strategies for problem solving: breadth-first and depth-first. Breadth-first problem solving might be thought of as funnel, while depth-first problem solving is more like a tunnel. Breadth-first involves delaying commitments, while depth-first involves making early commitments. Some people prefer the breadth-first approach, while others prefer the depth-first approach. However, most people prefer to use depth-first when approaching new problems, because this approach tends to quickly reduce the complexity of the problem to be solved.[23] Since design is, by definition, the consideration of a new problem, most novice designers are biased toward the depth-first approach.

The risk of depth-first problem solving is that the field under consideration will be narrowed too soon, especially if those making the early commitments are not experts in the domain. If a change of course is necessary, the work done in exploring the details will be lost, so this approach has a large cost of change.

Notice that both breadth-first and depth-first approaches require expertise in the domain. A depth-first approach will work only if there was a correct selection of the area to zero in on. Getting this selection right requires two things: someone with the expertise to make the early decisions correctly and assurance that there will not be any changes that render these decisions obsolete. Lacking these two conditions, a breadth-first approach will lead to better results.

A breadth-first approach requires someone with the expertise to understand how the details will most likely emerge and the savvy to know when the time to make commitments has arrived. However, the breadth-first approach does not need a stable domain; it is the approach of choice when the business domain is expected to evolve. It is also an effective approach when the domain is stable.

---

23. See Thimbleby, “Delaying Commitment,” 84.

### Personality Types

We, the authors, exemplify the breadth-first and depth-first personality types.[24] Tom has a strong bias toward delaying commitment, so he enjoys the process of evaluating options, sometimes at the expense of getting things done. Mary, on the other hand, in her eagerness to make things happen, quite often sets off down the wrong path. Since we have complementary strengths and weaknesses, we have learned how to combine them to get the best of both worlds.

When a decision must be made, it falls naturally to the person whose style is most appropriate. For example, Tom evaluates available computer networking approaches, while Mary decides when and what to buy. Tom mulls over the best approach for a new Web site, while Mary oversees getting the Web site developed and deployed.

Mary has learned that for important decisions, the results are always better if she delays commitment until Tom does the breadth-first search. Tom finds that Mary has developed a better sense of when decisions have to be made and is more likely to make things happen. However, when it comes to implementation, both know that pair troubleshooting resolves network issues and Web site problems much faster than working alone.

***—Mary and Tom***

## Intuitive Decision Making

Gary Klein studied decision making of emergency responders, military personnel, airline pilots, critical-care nurses, and others, to see how they make life-and-death decisions. He expected to find that these people make rational decisions in life-threatening situations; that is, they survey a range of options and weigh the benefits and risks of each option, then choose the best one from the analysis. When he started the study, he was amazed to discover that fire commanders felt they rarely, if ever, made decisions. Fire commanders were very experienced, or they would not have their jobs. They claimed that they just *knew* what to do based on their experience; there was no decision making involved. We call this *intuitive* decision making.[25]

When experienced people use pattern matching and mental simulation to make decisions, they are employing a very powerful tool that has an unquestioned track record of success. To make even better decisions, emergency responders, pilots, and military commanders engage in situational training that establishes correct pat-

24. Thimbleby, "Delaying Commitment," 84, sidebar.

25. Klein, *Sources of Power*, Chapter 3.

terns and enables better mental simulations. With the proper training and experience, intuitive decision making is highly successful the vast majority of the time.

Klein found that firefighter commanders resort to rational decision making only when experience is inadequate. Deliberating about options is a good idea for novices who have to think their way through decisions. However, intuitive decision making is the more mature approach to decisions, and it usually leads to better decisions as well.[26]

Rational decision making involves decomposing a problem, removing the context, applying analytical techniques, and exposing the process and results for discussion. This kind of decision making has a place in making incremental improvements, but it suffers from tunnel vision, intentionally ignoring the instincts of experienced people. It helps clarify complicated situations but contains significant ambiguity. Even though rational analysis gives specific answers, these are based on fuzzy assumptions and it is difficult to know exactly when and how to apply the rules.[27]

It would be nice if rational analysis could be counted on to point out when there is an inconsistency, when there is a key factor that everyone is overlooking. However, rational analysis is less useful than intuition in this regard, because rational analysis tends to remove context from analysis. Thus, rational decision making is unlikely to detect high-stakes mistakes; intuitive decision making is better in this regard.

Sometimes it seems that there are not enough experienced people available to allow intuitive decision making, and therefore rational decision making is the better approach. We strongly disagree. It is much more important to develop people with the expertise to make wise decisions than it is to develop decision-making processes that purportedly think for people. We are also convinced that it is quite possible to develop many people who are able to make wise intuitive decisions. Consider the Marines.

## The Marines

The U.S. Marine Corps doesn't have any real need to exist; the army, navy, and air force are equipped to handle any job the Marines tackle. However, the Marines specialize in chaos. "Everything about the Marines…is geared toward high-speed, high-complexity environments," writes David Freedman in *Corps Business*.[28] "The Marine Corps is one of the most open-minded, innovative, knowledge-oriented…organizations in the world."

---

26. Ibid., 23, 28–29. Note that intuitive decision making can yield incorrect results if the underlying assumptions are incorrect or the constraints are not understood.

27. Ibid., Chapter 15.

28. Freedman, *Corps Business*, xix.

Freedman outlines 30 management principles that the Marines use to enable young recruits to deal with extremely challenging combat missions as well as tricky, ill-defined humanitarian missions. If you want to know how to deal with complexity, the Marines have a few good ideas.

Marines plan, but they do not predict. A mission plan is both rapid and thorough, but it is not a scenario of how the mission will unfold. Instead, the planning process focuses on understanding the essence of the situation and the strengths and weaknesses of both sides; finding simplifying assumptions, boundary conditions, and alternate approaches; settling on an approach with a 70 percent chance of success; searching for what is being overlooked; and inviting dissent. These issues are covered rapidly in the hours immediately preceding the mission, and the Marines have a plan.

Once engaged in a mission, the organizational structure collapses, and those on the front lines, who have access to the most current information, are expected to make decisions. They also are expected to make mistakes. The theory is that they will make fewer, less serious mistakes than will distant officers. Mistakes are not penalized; they are considered necessary to learn the boundaries of what works and what doesn't.

Extreme training is used to be sure Marines don't encounter situations on the job more challenging than those faced in training. They develop skills and learn patterns that they are expected to adapt to new and changing situations. Training is done with stories and analogies, but Marines are not told how to do a job. Instead, the Marines "manage by end state and intent. [They] tell people what needs to be accomplished and why, and leave the details to them."[29]

The Marines focus attention and resources on small teams at the lowest levels of the organization. There is no personnel department; hiring, training, and assigning people are required and prized rotational assignments for every senior officer. They look for leaders who can motivate people, and they clearly distinguish management tasks—getting the maximum value from the dollar—from leadership tasks—helping people to excel. Marines are taught to be comfortable with paradox and value opposing traits. Thus, they learn to balance discipline and creativity, empowerment and hierarchy, plans and improvisation, rapid action and careful analysis.

Organizations that deal successfully with complexity, as do the Marines, understand that complex problems can be dealt with only on the front line. Thus, they focus on enabling intelligent, self-organizing, mission-focused behavior at the lowest levels of the organization. Marine leaders are trained how to clearly communicate command intent so that frontline people understand the mission and know how to make intuitive decisions.

---

29. Ibid., 208.

A clear statement of intent is the key to enabling emergent behavior on the front line. In business, the communicating intent is generally done through a small set of well chosen, simple rules.

## Simple Rules

Termites build amazing mounds and bees build complex hives. Birds migrate in flocks and fish swim in schools. These are not extraordinarily intelligent animals, yet as a group they exhibit extraordinarily sophisticated behavior. How do they know how to do it?

In the article "Swarm Intelligence" Eric Bonabeau and Christopher Meyer describe how ants find food efficiently by following two simple rules: (1) Lay down a chemical as you forage for food, and (2) follow the trail with the most chemical. If two ants go out looking for food, the one returning first will have laid down a double layer of chemical, so other ants will follow that trail, adding more chemical. The ants converge on the food very efficiently.

It turns out that these same two routing rules are equally effective in routing telephone traffic on a network. In this scheme, digital "ants" roam through a network, laying down "digital chemicals" in places of low congestion. Phone calls follow and reinforce the "digital trails." If congestion develops, the digital chemicals decay and are not reinforced, so calls are no longer attracted to that route.

Simple rules can lead to surprising results. Southwest Airlines had a rule that freight was to be loaded onto the first plane going in the right direction. The result was severe bottlenecks in the system even though very little of the overall capacity was being used. Then Southwest changed this one rule to ant forging rules: Find and use uncontested paths, just like the telecommunications industry. This meant that some cargo might actually start out moving away from its destination or taking a longer route than seemed necessary, which seemed very counterintuitive. However, the result was an 80 percent reduction in transfer rates at the busiest terminals, a 20 percent decrease in workload for cargo handlers, less need for cargo storage, and spare room on flights available for new business. Southwest estimated an annual gain of more than $10 million.[30]

Social insects act without supervision, self-organizing based on a set of simple rules. Their collective behavior results in efficient solutions to difficult problems. Bonabeau and Meyer call this *swarm intelligence* and list its advantages:

30. Bonabeau and Meyer, "Swarm Intelligence," 108.

- **Flexibility**—the group can quickly adapt to a changing environment.
- **Robustness**—even when one or more individuals fail, the group can still perform its tasks.
- **Self-organization**—the group needs relatively little supervision or top-down control.

Simple rules are very efficient at fostering flexibility, robustness, and self-organization in business environments. In the article "Strategy as Simple Rules," Kathleen Eisenhardt and Donald Sull note that managers have three choices when deciding how to compete. First they can "build a fortress and defend it." Second, they can count on unique resources to maintain a competitive advantage. Third, they can place their organizations in a position to rapidly pursue fleeting opportunities by choosing "a small number of strategically significant processes and [crafting] a few simple rules to guide them."[31]

The interesting thing about simple rules is that they enable decision making at the lowest levels of an organization. People do not have to wonder what to do in a situation or get permission to act. If they follow the simple rules, they know how to make decisions, and they know their decisions will be supported. Eisenhardt and Sull suggest that a simple-rules strategy gives a company a strong competitive advantage in high-velocity markets because they allow an entire organization to act uniformly and quickly with little supervision.

This is the key to simple rules: They allow everyone in an organization to act quickly, synchronously, in a coordinated manner, without the necessity of waiting for instructions from above. In a complex and changing environment, long decision chains slow decision making down and separate decision making from execution. Simple rules allow decisions to be made on the spot, when and where they need to be made, taking current information into account. Thus, simple rules are a key mechanism to enable people to *decide as late as possible*.

## Simple Rules for Software Development

Simple rules for knowledge workers are a bit different than simple rules for moving freight or switching packets through networks. Simple rules give people a framework for making decisions; they are not instructions telling people exactly what to do. Thus, simple rules are principles that will be applied differently in different domains; they are used by experienced people as guidance when making intuitive decisions.

31. Eisenhardt and Sull, "Strategy as Simple Rules."

People have a limit to the number of things they can consider when making a decision, so the list of simple rules should be short. George A. Miller's law suggests that somewhere around seven would be a good number.[32] Simple rules should be limited to the few key principles that really must be considered when making a decision. Quite often, simple rules are used to reinforce a paradigm shift, so they often focus on the counterintuitive elements of decision making.

We offer seven simple rules, or principles, for software development along with the tools to help you translate the principles to agile practices that are appropriate for your particular environment:

1. **Eliminate waste:** Spend time only on what adds real customer value.
2. **Amplify learning:** When you have tough problems, increase feedback.
3. **Decide as late as possible:** Keep your options open as long as practical, but no longer.
4. **Deliver as fast as possible:** Deliver value to customers as soon as they ask for it.
5. **Empower the team:** Let the people who add value use their full potential.
6. **Build integrity in:** Don't try to tack on integrity after the fact—build it in.
7. **See the whole:** Beware of the temptation to optimize parts at the expense of the whole.

A set of simple rules are guideposts; their purpose is to allow the people on the ground to make quick decisions about how to proceed, knowing that their decisions will not be second-guessed because they are making the same decision their managers would make given the same circumstances. It is the power that simple rules give to the people who add value that makes them so valuable. It is not so important that the rules give detailed guidance; it is important that people know that these rules are guidelines, which gives them the freedom to make their own decisions.

32. Miller, "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information."

## *TRY THIS*

1. Think of examples in your life when you have used options to delay decisions. For example, have you ever paid extra to lock in a low interest rate as you negotiated a mortgage? How effective has this been for you? Fill in the following table:

| Example of keeping options open | Very favorable result | No gain; lost the cost of the option | Very unfavorable result |
|---|---|---|---|
| Mortgage Negotiation | X | | |
| Example 2 | | X | |
| Example 3 | X | | |

   We think you will find most examples fall into either the favorable or no-gain category, but few fall into the unfavorable category.
2. At a team or department meeting, ask people to list decisions that are about to be made. Group the list of decisions into two categories—tough to make and easy to make. Then discuss what information you would need to turn each tough decision into an easy decision. Pick three tough decisions and apply the delaying tactics under "Tool 8: The Last Responsible Moment" to delay those decisions as long as possible.
3. Evaluate your personality—are you inclined toward breadth-first or depth-first problem solving? Find someone who has the opposite inclination, and pair with him or her as you decide how to approach your next development project.
4. Select a few critical processes and develop simple rules for them so that people understand intent and can make independent decisions.

*This page intentionally left blank*

# *Index*