CHAPTER 1

A representative sample of the book's contents

from

Flexible Product Development

by

Preston G. Smith

1

UNDERSTANDING FLEXIBILITY

> It is not the strongest of the species that survives, nor
> the most intelligent, but the one most responsive to
> change.
>
> —*Charles Darwin*

Flexibility has existed in industry for years, but the emphasis has been different from what this book addresses. To sharpen perceptions of the many flavors of flexibility, I start, a bit negatively, by mentioning some types of flexibility that are *not* my focus here.

Manufacturing professionals have embraced flexibility for a couple of decades. A technique from Toyota called "single-minute exchange of die" (SMED) has allowed them to change manufacturing tooling, such as the dies used to stamp body panels, in minutes rather than days, thus changing from making one part to making another one far more easily. Similarly, flexible machining centers allow manufacturers to change from machining one part to machining another one instantly, which further enables them to move from making one product to making another with little changeover cost.

Going further, product development professionals have moved these lessons upstream by designing families of products (sometimes called platforms) that allow so-called *mass customization,* that is, the ability to adapt a product late in manufacturing—or even in distribution—to meet the needs of an individual consumer.

This is a popular view of product development flexibility, but it is not what I address here. Instead, I'm talking about the changes that occur during the process of *developing* the product. In particular, this book addresses a growing conflict between so-called best

practice in product development—which says that one should plan the development project and then follow the plan—and the reality of today's industrial environment—where change from original plans is the norm, not the exception. I discuss this conflict more in the next section, but at this point it's useful to define the type of flexibility covered in this book:

> *Product development flexibility*: The ability to make changes in the product being developed or in how it is developed, even relatively late in development, without being too disruptive. The later one can make changes, the more flexible the process is. The less disruptive the changes are, the more flexible the process is.

You might be tempted to create such a mathematical formula: flexibility equals time-into-project times lack-of-disruption. Unfortunately, it is more complicated, because each change is different and leads to a different amount and kind of disruption. One change might affect marketability of the product, while the next might waste development labor. Furthermore, a change that is harmless in one phase of the project could be disastrous in the next.

Dealing with Change

Change is fundamental to product innovation, which, after all, is about bringing something into being that hasn't existed before. *The more innovative your product, the more likely you are to make changes during its development.*

Recent studies show that innovation connects strongly with long-term corporate success, and corporate executives regularly list innovation as a top critical success factor. For example, one global survey ranks innovation as the top strategic priority for 40 percent of senior executives and among the top three strategic priorities for 72 percent of these executives.[1] Nevertheless, research shows that corporate product portfolios are becoming less innovative. See Figure 1.1, which shows that over a fourteen-year period and over a broad

Figure 1.1 Decline in Innovation, as Shown by Portfolio Shift: 1990–2004
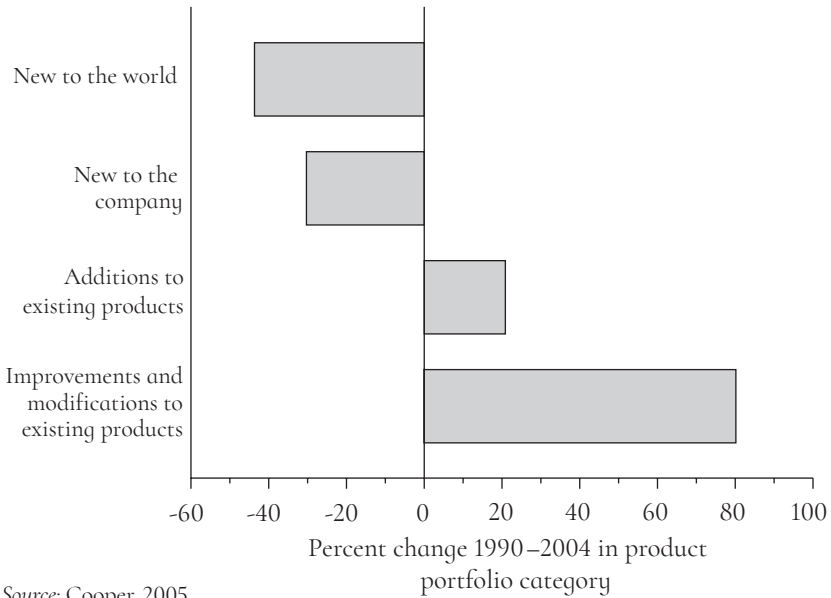
Source: Cooper, 2005.

range of industries, the proportion of truly innovative products in corporate portfolios has decreased while the proportion of simple upgrades in portfolios has increased. In short, innovation is vital to business success, but contemporary businesses are losing the innovation battle.

Why is this? Many possible explanations come to mind, but I believe that it is due to competitive pressures and a short-term outlook forced upon executives by the financial markets. Today's executives simply cannot afford to be wrong. Seeing the great advances achieved in the factory by driving variation out of the manufacturing process, they also want to reduce product innovation to a predictable activity. This also explains the current great interest in Six Sigma, which is a methodology to drive variation out of all parts of the business.

Six Sigma, ISO 9000, and similar quality systems are not the only culprits. Stage-Gate, PACE (Product and Cycle-Time Excellence), NPI (New Product Introduction), PDP (Product Development

Process), and similar phased product development systems encourage heavy up-front planning followed by sticking to the plan. And you can add to these the phenomenal recent growth of project management,[2] project office, and similar methodologies that promote a plan-your-work, work-your-plan approach.

None of these approaches is misguided or has net negative effects. When applied to high-risk, highly innovative programs, however, they have had an unnoticed side effect of putting innovation in a straitjacket, thus making it increasingly difficult to make changes in projects midstream in development. Those who must make such changes are often penalized and regret what they are doing—when, in fact, they are innovating. For stable projects, the current trends of greater planning and control are properly aligned, but for the more volatile ones aimed at correcting the portfolio problem illustrated in Figure 1.1, developers need more flexibility to make midstream changes.

What kinds of changes are these?[3] The first group is changes in customer requirements. Often, customers must see the actual product before they can relate to it, the IKIWISI (I'll know it when I see it) phenomenon. Sometimes they have unanticipated difficulties in using early versions of the product or they try to use it in unanticipated ways. Often they find completely new uses for or ways of using a product. Software developers call these *emergent requirements*, because they emerge in the course of development and no amount of market research is likely to uncover them in advance. Occasionally, features introduced by competitors drive changes. As customers or developers try prototypes, they discover better or cheaper ways of delivering the specified customer benefits.

Related are market changes. Competitors come into being or go out of business. They introduce unanticipated and disruptive products. Markets change in response to fads, shifts in customer preferences, government or regulatory action, or political events. Often, markets are new and thus poorly understood; for instance, it took 3M, manufacturer of Post-it brand sticky notes, nearly a decade to find a market for this item, which is indispensable today.[4]

Then there are technology changes. Sometimes a new technology does not work as advertised. Or it may work better than expected, and developers want to exploit this. It may have unexpected side effects or require additional work to render it acceptably reliable or user friendly. Sometimes patent infringement or licensing problems arise.

World events, such as terrorism or global warming, can lead to changes in a development project.

The next group might be called network changes. Seldom do companies today develop a product entirely by themselves. Sometimes they engage consultants who are expert in a particular area. Often a supplier provides components. Increasingly, partners develop subassemblies from general guidelines. Complicating this, these partners tend to be located in distant parts of the world. Such broad and dispersed networks are fertile ground for changes. For instance, a supplier receives a big, urgent order from another customer—maybe even your competitor—that compromises your order.

Finally, there are organizational changes. Managers are promoted or reassigned. Key employees leave. Managers move developers from project to project to resolve changing priorities. Project budgets are cut. Management lets some initiatives wither while starting new ones.

What can you do about these changes? You may have other options, but I see three. First, you can move faster to minimize your exposure to change. This is an approach taken in agile software development, and one I have recommended in the past.[5] Agilists divide development into short iterations, typically of one to four weeks each. Then they can freeze the plans during an iteration while replanning between iterations. Broadly speaking, rapid development techniques rely on working quickly enough to avoid change during development.

 Second, you can plan better in hopes of anticipating change. This is the approach followed by Six Sigma, phased development, and similar techniques mentioned earlier that emphasize more

formal customer research, more structured risk management, and similar up-front work. This book also develops anticipation as a flexibility technique in certain cases. But the amount of anticipation possible is severely limited. An analogy here is the needle-in-a-haystack metaphor. Imagine building up a haystack by tossing more hay on top. Then, as the stack grows in all three dimensions at once, the difficulty of finding a needle in the midst of all that hay (that is, anticipating a certain change) is proportional to the third power of the haystack's height, which is analogous to how far in the future you are forecasting. See Figure 1.2.

This is an apt metaphor. Real-world projects change in several dimensions at once, so anticipating a certain change becomes increasingly difficult as one tries to extend the forecasting horizon. Eventually, the work put into planning to anticipate change reaches a point of diminishing returns.

Third, you can build a process and apply tools and approaches that are more tolerant of change—ones that accommodate and even embrace change as a natural consequence of working in the

Figure 1.2 Difficulty of Finding a Needle in a Haystack
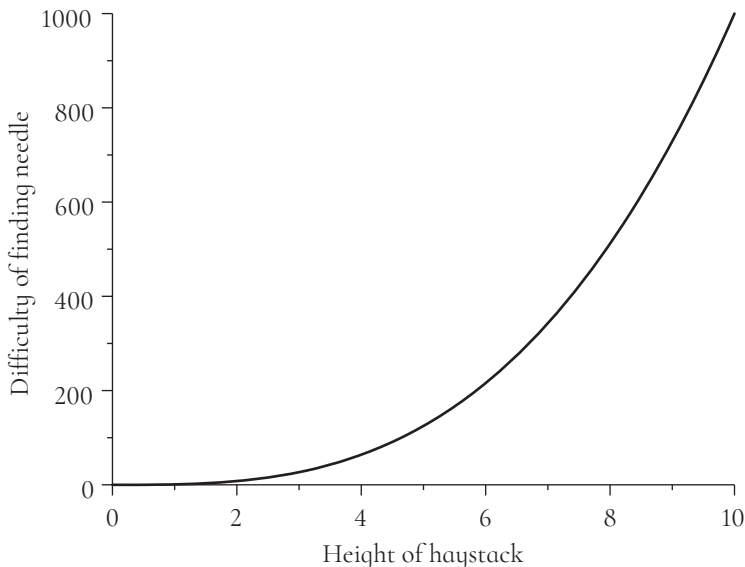
innovative domain where change is the norm. This is the direction I offer in this book. Change can have great opportunity associated with it, and it is better to exploit that opportunity than to suppress it.

How Much Flexibility?

From the definition of flexibility provided earlier, it would be easy to conclude that the more flexibility, the better. But flexibility can be expensive, so it must be used with discretion. This is an area where cost-benefit thinking pays off.

Benefits of Flexibility

The benefits of flexibility connect directly with the degree of innovation you seek. Figure 1.1 suggested that new products could benefit from more innovation, and this is generally true. But new-product experts also agree that a new-product portfolio needs balance between innovative products and line extensions.[6] In general, mature products provide reliable income for today; innovative ones ensure that you will be in business tomorrow.

Consequently, apply flexibility where you must be innovative. You can do this at several levels:

- Some markets for your products change faster than others.
- Some product lines within a company change more than others.
- Some products within a line are more subject to change than others.
- Some portions of a product are subject to greater change than others, due to immature technology, unstable customer needs, or market flux.
- Some departments or disciplines change faster than others, for example, the electronics in an airplane change much faster than its structure.

This is where competitive advantage resides—in distinguishing where the organization is going to pursue innovation and thus needs flexibility, and where it will encourage stability and its associated economies. No simple rules apply here, but these decisions should stem from your corporate strategy and from an understanding of the uncertain spots in your technologies and markets.

After this chapter, this book provides tools and approaches for enhancing flexibility. You should decide where and to what degree to apply them. Use discretion, but remember Figure 1.1: you probably could benefit from considerably more flexibility than you have today.

The Cost of Change

Managers resist change in a project—quite correctly—because it is expensive, and change usually leads to schedule slippage. Furthermore, change can open the door to product defects. So any attempt to encourage change must consider its cost. Although each change has different effects on the project, the cost of a change, in general, rises the later it occurs in the project. Barry Boehm has collected data for the cost of fixing an error in a large software program, averaged over many large projects from TRW, IBM, GTE, and Bell Labs. As shown in Figure 1.3 , the cost rises exponentially by a factor of 100 from the requirements phase (cost: 1.6) to the operational phase (cost: 170). (Similar data for a few small, less formal software projects, however, indicate that these smaller projects only increase the cost of change by a factor of 5 from requirements to operation).[7]

More recent data from Boehm confirm that the factor of 100 still holds for contemporary large projects, and he has also found that the Pareto principle applies: 80 percent of the cost of change comes from only 20 percent of the most disruptive changes, namely, those with systemwide impact.[8] Furthermore, this group of expensive changes is usually identifiable in advance, and by applying the tools covered in this book, you'll find you can often avoid them. This, plus the

Figure 1.3 Cost of Changing Software
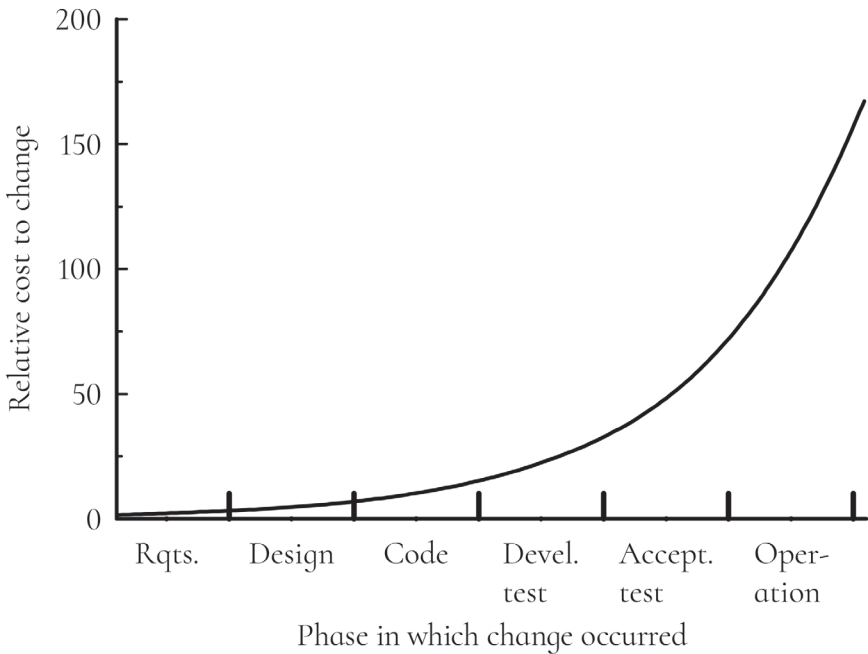
Source: Boehm, 1981, p. 40.

fact that the cost of change is lower for small projects, is very good news. Product developers can take advantage of both these opportunities.

Earlier I suggested applying flexibility selectively and at a level where you believe change is most valuable or likely. Now another criterion appears for selecting the areas where you wish to be flexible: avoid the areas with systemwide impact—the ones most likely to have a high cost of change (if you cannot resolve them by using the tools in this book).

A word of warning. The cost of change is a hotly debated topic among developers, usually based on their own undocumented experience or perceptions. As far as I know, Boehm's data provide the only carefully collected and documented information available. Boehm himself is highly regarded and has collected his data

from many sources over thirty years. You are likely to hear "rules of thumb" that the cost of change escalates by a factor of 10 for each phase of development, which would raise the Figure 1.3 factor to 100,000! Such factors seem to be pure conjecture. Please question your sources on the cost of change.

Also, notice that all of Boehm's data are for software projects, which unfortunately limits their application to other fields. I have thus far found only one limited source of cost-of-change data for mechanical, electrical, chemical, optical, or mixed systems, but I have little reason to doubt that Boehm's findings carry over in general.[9]

Usually, discussions about the cost of change revolve around a graphic like Figure 1.3 , but as the discussion here suggests, we know little quantitatively about the cost of change. Nevertheless, this is a valuable concept, because midstream changes do have associated costs that should be kept in mind continually and tied to the related benefits of flexibility. Many of the tools in this book aim at reducing the cost of change.

Managing the Convergence of Flexibility

Figure 1.4 shows three levels of flexibility after the initial planning period. The restricted flexibility level is the most common, as it fits with popular good practice in phased product development. At the outset, the project has complete flexibility, since nothing is fixed yet. But at the end of the initial (planning) phase, the project budget, schedule, and product requirements are established and approved. From here on, this project has restricted flexibility, as shown in the figure.

Next, consider the completely flexible zone in the figure. In this idealized case, the ability to make changes is left wide open until the end of the project. This yields lots of flexibility but also leads to chaos at the end of the project when nothing is yet certain. The project schedule will most likely stretch, as will the budget. Consequently, complete flexibility is not a useful objective.

Figure 1.4 Three Levels of Managing Flexibility in a Development Project
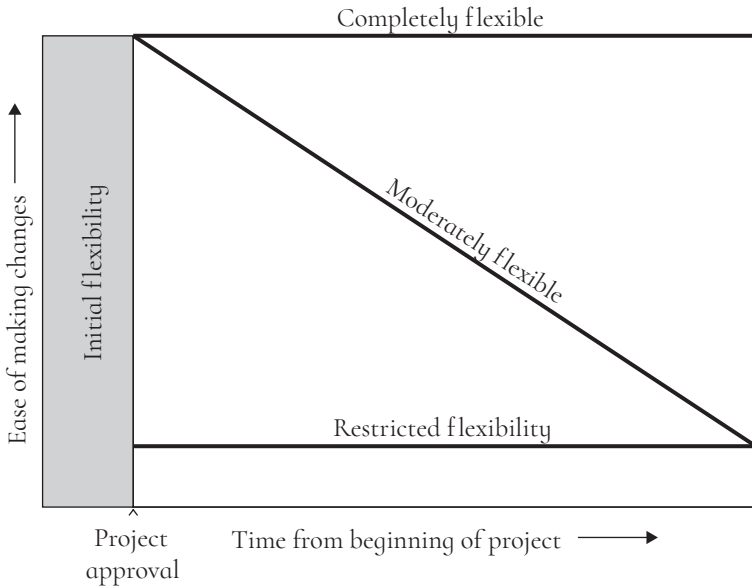
A project managed for flexibility will look like the moderately flexible zone in the figure. It starts with a great deal of ability to make changes. Decisions are not made until they must be made— what I describe in depth as the "last responsible moment" in Chapter Seven. But decisions are made when necessary, often by progressively tightening up tolerances on variables. Thus the ability to make changes narrows methodically as development proceeds. At Toyota, a major duty of engineering managers is to manage the rate of convergence of the design space: not so fast as to rule out change unnecessarily but not so slow as to leave too much uncertainty late in the project.

The Downsides of Flexibility

Flexibility has its place, which is in projects or portions of projects where change is likely to occur. However, this ability to accommodate change can be abused by managers who introduce unnecessary

change simply because the system is now more tolerant of it. Think of a high-performance motorcycle: it can get you to your destination quickly, but it can also get you into the hospital quickly. To survive, you must ride your motorcycle with skill and wisdom.

Similarly, an advantage of a flexible approach is that it can follow customer reactions quickly, but if the customer vacillates or is flighty, the project can become chaotic.

Flexibility can be a crutch for indecisiveness, for not committing to decisions, or for reversing prior decisions. Sometimes, flexibility is abused by those who do sloppy research or planning, thinking that a flexible system will allow their work to be fixed later. More broadly, it can be an excuse for skipping the planning and thus emphasizing firefighting and tactical views over a strategic view.

If you use flexibility as an excuse to be sloppy, you will derive no benefit from it.

The Roots: Agile Software Development

To my knowledge, no other books on flexible product development have been written, and only a few articles. However, non-software developers can draw on a rich body of material in a parallel field: agile software development. Although its roots go back further, agile development has arisen since about 2001. Its starting point is the Agile Manifesto (see Exhibit 1.1), which appeared in February 2001. The annual agile conference has grown in attendance from 238 in 2001 to 1,111 in 2006, a compound annual growth rate of 36 percent. More than fifty books and countless articles now exist on agile software development; Craig Larman provides a good over-view.[10] The remainder of this chapter provides some highlights of agile development to provide background for the non-software flexibility material covered in later chapters. Toward the end of this

---

Exhibit 1.1 Manifesto for Agile Software Development

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

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

That is, while there is value in the items on the right, we value the items on the left more.

Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas.

The Agile Manifesto. © 2001, the above authors. This declaration may be freely copied in any form, but only in its entirety through this notice. Source: agilemanifesto.org (Accessed August 31, 2018).

chapter, I discuss the differences between software and non-software development that prevent us from adopting the agile development approaches directly.

I refer to the Agile Manifesto more later, but for now, notice that it is four statements that contrast values. Although the second value of each pair is acknowledged as being valuable, the agilists emphasize the first value more. Also notice that the second values align closely with generally considered "best practice" in the traditional development of new products (*product development* can be substituted freely for *software development* in the manifesto). The themes of the Agile Manifesto pervade the agile approach to software development. Accompanying the manifesto is a set of thirteen principles that underlie it.[11]

In practice, agile software development is a collection of about seven identifiable methodologies:

- Extreme Programming (XP)
- Scrum
- Adaptive Software Development
- Crystal (a collection of methods for various types of projects)
- Lean Development
- Dynamic Systems Development Method (DSDM)
- Feature-Driven Development (FDD)

On a given project, developers often use a mix of these methods, a combination of XP and Scrum perhaps being most common.

More important than the differences between these methods is the commonality among them. They all follow the Agile Manifesto. They all develop software in short iterations—from one to four weeks, occasionally to six weeks. They all produce working software at the end of each iteration. They all deliver releases to customers frequently. They all involve the customer either directly or through a surrogate (such as a product manager), usually at the end of each iteration and sometimes on a daily basis. They all invite change at the end of each iteration (but essentially prohibit it during an iteration). They all tend to do planning and risk management as they proceed with the iterations. They all emphasize small, co-located teams. They all use emergent processes (those that emerge during the project, not determined at the beginning of the project). Observe that most of these characteristics can be adapted to non-software projects.

Extreme Programming

Extreme Programming (XP) is perhaps the most widely discussed and illuminating of the agile methodologies, so it's worth considering in detail.[12] It is based on a dozen or so practices that fall into the context of about four values. Although the values actually come

first, I will start right off with the practices. Once you absorb them, the values that create an environment in which the practices have a chance of working will make more sense.

XP Practices

I use the terminology of those who practice XP here so that you can become used to it, then switch to more conventional product development terminology later. I urge you simply to observe these practices now without concern about how they might ever apply to non-software products. This is the task of later chapters. However, do observe as you read these that many of them improve flexibility by lowering the cost of change.

The Planning Game. Some people accuse agilists of not planning. True, initial project planning may be light—because so much is likely to change before it is used—but this is more than compensated for by detailed planning within an iteration. This iteration planning is always a balance of business and customer needs against the team's capability and capacity. The business and customers lead in deciding on the features to be developed, their priorities, and the timing for a release (a release usually comprises several iterations). The developers lead in estimating how much effort a feature will require (and thus how many features can be completed in an iteration), the work processes the team will use, and detailed scheduling and prioritization within an iteration. The important factors are that developers plan iteratively and that there is strong interplay between the business or customer people and the technical people—and clear roles for them all.

Small Releases. The emphasis is on small and thus frequent releases to customers to enhance opportunities for feedback and flexibility. For instance, a company using these techniques for educational software used in public schools (not an industry subject to frequent change) plans releases every eight weeks. Clearly, this requires that the fixed

cost of a release, such as costs of documentation, training, and flushing the distribution channel of old products, be reduced.

Product Metaphor. This is a vision of the product, held in common by the team, that indicates what it will do or how it will differ from what exists now. It provides the team with a compass to know whether it is going in the right direction in the stormy seas of change. I cover product visions in detail in Chapter Two. Of all the practices of XP, this one has been the most difficult to implement. This may be because a metaphor cannot capture all products crisply, or it may be that it has not received enough attention, whether by the methodologists in describing how to create a metaphor or by the teams in allocating adequate time for creating a captivating metaphor.

Simple Design. This one runs counter to the way designers normally operate. It says that one should design and implement *only* what is necessary to satisfy today's requirement or what the customer needs today. The idea is that if the landscape is changing constantly, speculating on tomorrow's needs will most likely be wrong. Not only does this waste resources, it complicates the design and thus raises the cost of change for tomorrow's work. The agile term for this is "barely sufficient." Following the fourth item of the Agile Manifesto, it means that one places more value on adapting than on anticipating. Note that while this premise is appropriate for projects subject to a great deal of change, it is not wise for projects that are predictable. Also, it runs counter to the principle of providing reserve performance, which I cover in Chapter Three.

Test-Driven Design. Much like non-software development, software traditionally is developed in large batches of features. When development is complete, programmers turn their code over to a tester, who then designs tests to ensure that the features work properly and do not cause damage elsewhere. XP turns this

around by having the programmer, working a feature at a time, write the test first and then code the feature to pass the test, rather like being offered the final exam when you start a class. Among other things, this encourages simple design, because developers can code a feature in a barely sufficient way to pass the known test. In addition, they write all tests to be automated with a clear pass/fail outcome for each one, so that tests accumulate and can be run repeatedly to confirm that existing features still work as others are added.

Refactoring. This is a process of cleaning up code without changing its behavior. The cleanup could be to render it more understandable, to improve its internal consistency, to streamline its design or remove duplication, or to make it easier to work with in the future. Explicitly, refactoring does not add capability to the code. There is nothing basically new here: programmers have often cleaned up code as a first step in preparing to modify it. But in XP and other agile methodologies, refactoring is a routine activity done apart from adding new features and, indeed, whenever an opportunity to refactor appears. Agilists are meticulous about the cleanliness of their code, as this keeps the cost of change low. Notice that the automated tests just mentioned are a prerequisite for refactoring, because the developer must run the refactored code through the test suite to confirm that its behavior hasn't changed.

Pairing (*Pair Programming*[13]). XP requires that all production code be written by two programmers sitting at one computer with one keyboard and one mouse. One of them, called the driver, enters code while the other, the navigator, plays various roles as needed: strategist, checker, planner of next steps, or contrarian. They operate as equals, and they trade roles a few times every hour and change partners once or twice a day. Although you might assume that this would double labor costs, several studies have shown that it adds 10 to 15 percent to costs while reducing defects by about 60 percent and shrinking schedules by about 45 percent.[14] A major benefit

is that it gives everyone a shared understanding of all of the code, which sets us up for the next practice, collective ownership.

Collective Code Ownership. This means that the entire team owns all the code, and anyone on the team has the right to change any of it at any time—in fact, the obligation to refactor it if an opportunity to do so appears. Clearly, this could lead to errors, but pairing and continuous automated testing protect against undesirable results. A major advantage of collective ownership is that the code is not hostage to the specialist who created any particular part of it, again lowering the cost of change.

Continuous Integration. Pairs may be working with their version of the code, but they frequently integrate it with the common version on the server and run all the automated tests immediately. Then they discover quickly if they have broken the code (that's *fast feedback*, an important theme of this book). Clearly, the beauty of this is that problems surface quickly and clearly compared to the normal situation, where integration happens infrequently, obscuring the fault. Notice that continuous integration takes advantage of modern technology (fast computers and easy-to-use integration software) for process advantage, which is another theme of this book.

Sustainable Pace. All agile methods are people-oriented. The authors of the Agile Manifesto were no strangers to the burnout— or *death march*, as one popular book on software development is titled—that accompanies too many software development projects. Although this concept does not appear in the manifesto itself, it is explicit in the published principles behind the manifesto. The rule in XP is clear: if you work overtime one week, you can't work overtime the next week. The thinking is that if a problem requires two consecutive weeks of overtime, more overtime will not fix it.

Customer on the Team. We all recognize the value of having access to a customer when detailed questions of usage surface or

priorities must be set under limited resources. Again, XP goes to the extreme. The rule is that a real customer must sit with the team. As you might guess, teams sometimes sidestep this, but it stands as the XP rule nevertheless. Observe that many software projects are IT (information technology) ones done for a customer who is within the organization, for example, order-taking software for the company. Thus, for many XP projects, it is easier to identify and assign a customer than it would be for many non-software projects.

Coding Standards. The practice of maintaining coding standards supports other practices, such as collective ownership, pairing, and refactoring. With so much built-in fluidity (to provide flexibility), the team simply cannot allow a laissez-faire approach to formatting, style, and similar matters. It should be impossible to tell who wrote which part of the code. The team can establish its own standards or it can assume them from company standards or those supplied by the software language in use. Common standards are one strength that Toyota uses to remain flexible much deeper into the development process than its competitors. Viewed another way, by standardizing things that normally remain constant, you gain latitude to let variability run longer in the design itself.

It is important to recognize that the practices do not stand alone. They fit together and support one another mutually. There is a worn story about two programmers meeting and one says, "We are doing XP."

The other asks, "Are you doing pairing?"

"No."

"What about test-driven design? Are you doing that?"

"Nope."

After a couple more rounds, in frustration, the first programmer asks, "What *are* you doing?"

"We've stopped doing the documentation."

Clearly, effective use of XP goes beyond eliminating obnoxious activities; it must extend to recognizing that these practices are designed to fit together like a puzzle to provide a safety net for

one another. For example, see Figure 1.5, which shows the support structure for one practice. Similar support is possible for all the other practices as well.[15]
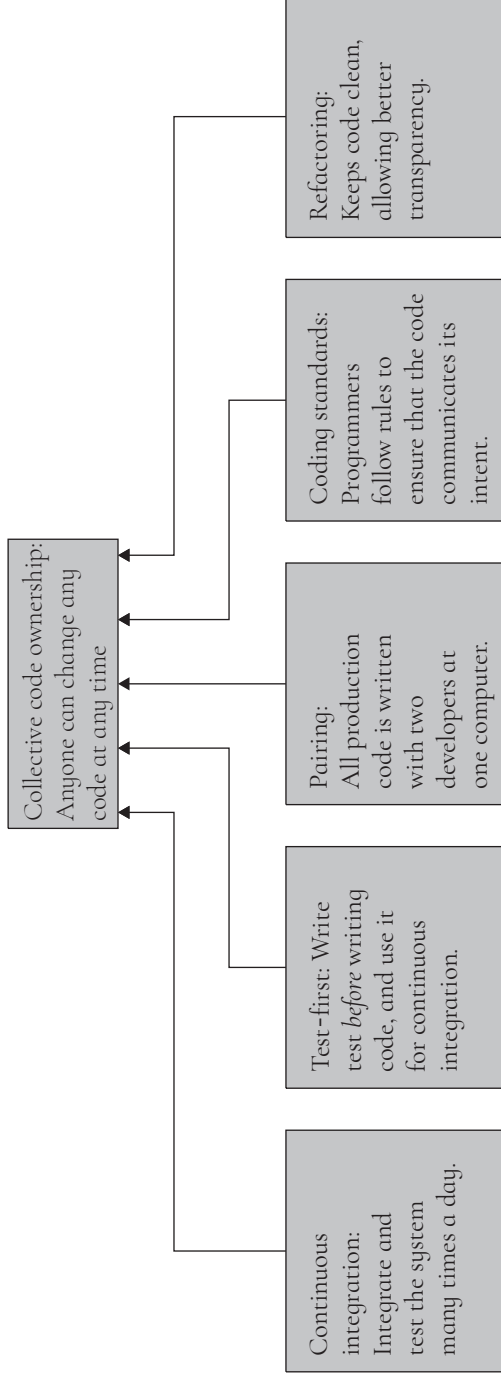
How Did XP Arise?

The story about how XP started is instructive.[16] Kent Beck, one of the originators of the technique, was consulting at Chrysler in 1996 when he was asked to lead a programming team. He observed what seemed to be best practices and asked the team to do them, things such as testing early in the project and co-location. The next time, there was more at stake and he was under pressure, so (as he told an interviewer later), "I thought, 'Damn the torpedoes' … [and] asked the team to crank up all the knobs to 10 on the things I thought were essential and leave out everything else." So, for example, testing early seemed to be a good idea, so why not write the test *before* writing the code? Co-location seemed to be beneficial, so why not put two programmers side by side? Thus, XP was born. What are you doing today that is working and might be cranked up to 10?

XP Values

Extreme Programming may not be completely transferable outside the software development world, but its underlying values are. What are they? Beck lists four: communication, simplicity, feedback, and courage.[17]

   Communication. Communication is at the very core of much of human activity, and nowhere more so than in product development. Good communication is difficult. We forget to tell people things that are critical to their work, or we are unaware that this information is critical to their work. Sometimes we hide information purposely, because it is embarrassing. On occasion, people ignore what we are saying or they miss it because they are distracted by something else.

Figure 1.5 "Safety Net" for Collective Code Ownership

Collective code ownership: Anyone can change any code at any time

Continuous integration: Integrate and test the system many times a day.

Test–first: Write test *before* writing code, and use it for continuous integration.

Pairing: All production code is written with two developers at one computer.

Coding standards: Programmers follow rules to ensure that the code communicates its intent.

Refactoring: Keeps code clean, allowing better transparency.

When we are working in the flexible mode, another dimension enters: communicating the certainty or flexibility in your basic statement, for example: "I would like to have ten units by Friday so that I can test them over the weekend, but next Tuesday is my absolute deadline. And if you can't get new ones at a decent price, functional used ones will do."

Another factor that enters these days is the effect of cultural factors on communication. For example, I do some training for a Chinese training company, and I recently went with them to conduct a workshop in India. The Chinese trainers were amazed at the contrast between the two countries. In China, the students are reticent, and we spend our time drawing them out and seeding a discussion. In the Confucian manner, subordinates defer to the boss. In India, it is the opposite: the discussion expands and we must strive to manage it and bring it to closure. Each participant has an opinion that must be expressed.

Simplicity. In XP, simple design and refactoring aim directly at simplicity. The idea is that something simple is easier than something complex to understand and thus to change. Complexity hides problems and extends the time needed to understand how something works and would work if you were to change it. In other words, simplicity lowers the cost of change. As Albert Einstein put it, "Everything should be made as simple as possible, but not simpler." The principles provided with the Agile Manifesto describe simplicity as the art of maximizing the amount of work not done.

Simple design is a knobs-at-ten approach to simplicity that, as noted earlier, does not always apply. However, the value of simplicity is much broader than this. Most of us overload ourselves when we travel with clothing that we never wear, we cover our desks with piles of paper that have outlived their utility, and we carry older, low-volume products in inventory just in case someone wants them. Simplicity is about clearing out these things in order to be clearer and more adaptable where it matters today.

Feedback. Feedback drives flexible systems better than plans do. The fourth value of the Agile Manifesto is "responding to change over following a plan." Note that half of the XP practices exploit feedback:

- The Planning Game, to plan the current iteration based on what you learned from the preceding one
- Small Releases, so that you can learn early what the marketplace thinks
- Test-Driven Design, so that you discover quickly whether your work is right
- Pairing, to correct incorrect thinking even earlier
- Continuous Integration, to learn sooner whether you have problems
- Customer on the Team, to know what the customer thinks as early as possible

Courage. When Kent Beck said "Damn the torpedoes" and cranked the knobs up, he didn't know if it would work. That took courage. Several of the XP practices require courage. One is simple design—purposely not putting in the design what you might need tomorrow. It takes courage to refactor code that someone else wrote, possibly causing it to break. Pairing takes courage; it is a good way to get your ego bruised.

Observe that courage is supported by the other values of communication, simplicity, and feedback. Good communication gives you the best information for taking action, so you have the best chance of success. Simplicity allows you to see through the haze, further raising your chances of success. And feedback allows you to revise and redirect quickly if you are wrong.

Does XP Work?

Extreme Programming is certainly a radical departure from traditional "best practice" in software development. What is its record

of accomplishment? You certainly can find naysayers who trot out failures, and you can find plenty of advocates with wonderful success stories. Perhaps the most even-handed assessment is reported by Boehm and Turner, who advocate a balance between agile and traditional methods.[18] They found that for thirty-one agile (mostly XP) projects in several industries (aerospace, computer, and telecom, among others), compared to traditionally run projects of similar complexity:

- All were average or above average in budget performance.
- All were average or above average in schedule compliance.
- All were roughly the same as traditional projects in product quality.

They also observed that these projects followed most XP practices strongly, but that a full-time co-located customer and a forty-hour week were admirable aspirations but difficult to achieve in practice.

The big advantage of these methods for us is in flexibility. During each iteration (one to four weeks), the set of features being implemented is replanned, so

- Customers or marketing people can add a new feature in under a month with no penalty.
- They can drop a feature that hasn't been processed yet with no penalty.
- Management can terminate the project at any time with the most valuable features completely coded and tested.

However, XP has been used mostly on smaller projects where it was possible to have a co-located team in one room (I have much more to say about co-located teams in Chapter Six). Also many of these projects were internal IT ones where it was relatively easy to involve the customer. As experience with XP and other agile

methodologies is growing, however, these restrictions are loosening.[19]

Moving from Software to Other Products

Software is a special medium that lends itself to agile approaches. Here are some of software's characteristics that agilists have exploited:

- Object technologies, which allow modularization to isolate change and enable substitution of modules
- The low cost of an automated build, which facilitates frequent and early testing
- The logic basis of software, which allows relatively fast automated checking for many types of errors
- Relatively easy divisibility of product features, which enables developing a product feature by feature and subdividing features to split tasks
- (For IT projects) customers who are relatively easy to find and involve in development
- In general, the malleability of the software medium, which makes change relatively easy

Nevertheless, agilists have worked hard to exploit the special characteristics of software to their advantage. Many of the principles agilists exploit apply equally to non-software products, principles such as iterative development with customer feedback, self-organizing teams, and emergent processes. We can exploit the special characteristics of non-software media to our advantage. For example, an advantage mechanical systems possess is that they are quite visible, lending themselves to physical prototyping, and electrical systems have the advantage that programmable components, such as field-programmable gate arrays, allow quick changes in a system that may be difficult to redesign and rebuild quickly.

Consider how Johnson & Johnson Worldwide Emerging Markets Innovation Center (Shanghai) has translated many of the XP practices to their business of developing personal care products, such as lotions, creams, shower gels, and soaps:

- Small releases: Conduct fast prototypes and test immediately. They separate the development of fragrances, preservative options, and base formulas, and then merge them eventually.
- Simple design: Remove unnecessary materials in the formulation.
- Test-driven design: For example, when concerned about skin moisturization, look at how the test is done and design the product accordingly.
- Pairing: Adopt a buddy system. Have two formulators working on the same project, which helps both in finding better solutions and in broadening skills.
- Collective code ownership: For difficult issues, conduct group prototyping wherein a pair of buddies shares their issue with others, whereupon other pairs create solutions in the lab and forward them to the requesting buddies for further development.
- Continuous integration: As soon as formulators create an innovative product, forward it to others who optimize the formulation.
- Customer on the team: Expose fast prototypes to consumers and get their assessment, then revise, evaluate again, and so forth.

A Note of Caution

I have mentioned that these tools and techniques must be applied selectively to some projects and not to others, and they should be applied only to certain portions of a project. The next section provides more on this. Some of the tools and approaches, such as simple design, are exactly the right thing to do in some cases and

absolutely the wrong thing to do in others (see "Providing for Growth" in Chapter Three, for instance). I point out many of the potential pitfalls as I go along. I wish I could resolve the essential ambiguity for you, but this is impossible given the broad variety of potential applications.

This is new material. Tomorrow it will be applied in ways undreamt of today, and this will lead to clearer rules for when and how to apply it. You could wait for the material to be packaged neatly for you by your competitor, but that is probably an unattractive option.

Consequently, these tools and techniques should be applied by a seasoned manager who understands the unique objectives and capabilities of the target organization, as well as its culture and the demands of its marketplace.
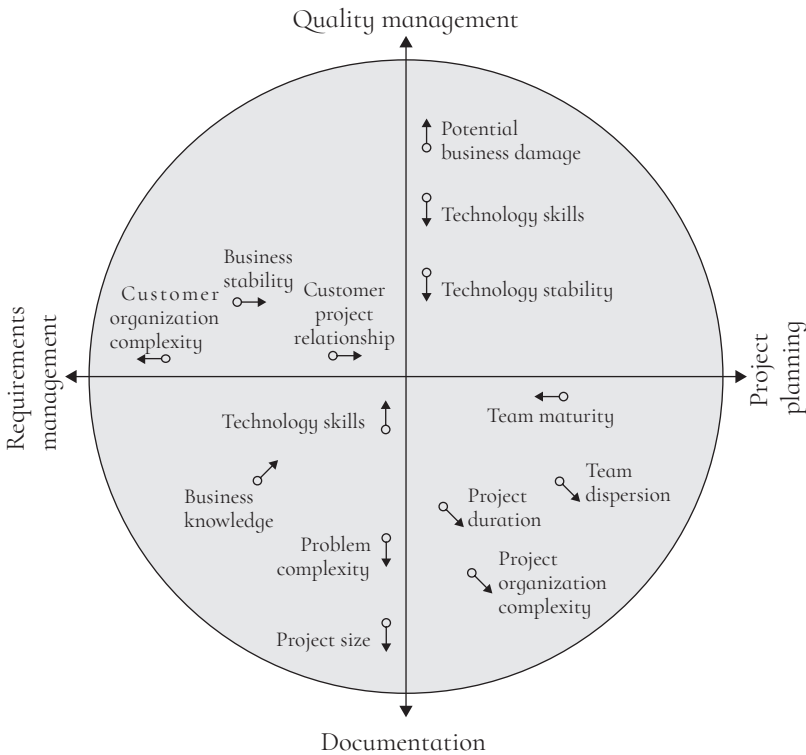
The Project Analyzer

Flexibility is inappropriate for some projects, nor is it necessarily appropriate for all parts of a product. The Project Analyzer suggests where to allow flexibility and where it is necessary to stay closer to more traditional approaches (See Figure 1.6). It measures a project in four dimensions: quality management, project planning, documentation, and requirements management. Each of these dimensions is independent and is influenced by various project attributes, as shown in Figure 1.6b.

As indicated in Figure 1.6a, a project can rate high on one axis, necessitating a greater amount of structure and control there. On the other hand, other dimensions might be relatively light, allowing more flexibility on them. For instance, in Figure 1.6a, Project 2 can be managed using a more flexible approach than Project 1—except in requirements management, where it should be even more structured than Project 1. Also note that the total "thumbprint" area of a project indicates the amount of project overhead required, and the area left over outside the thumbprint represents the residual effort, after overhead, for actual product development.

Figure 1.6 a. The Thumbprints of Two Projects.
b. Attributes of a Project That Push Its Thumbprint
Along the Indicated Axis

Project 1

Project 2

(a)

(b)

© 2002 Anne Bjerre Jorgensen and Ole Jepsen. Used with permission.
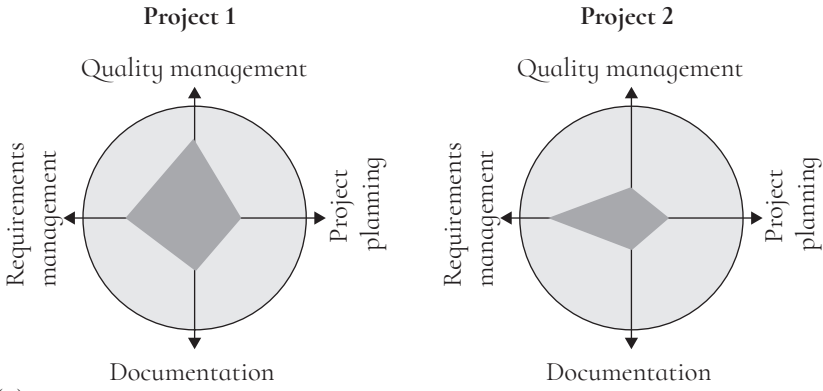
The point of Figure 1.6 (and my motivation for placing the Project Analyzer so early in the book) is only to alert you that each project should be viewed separately for areas where flexibility will be beneficial and where it might be harmful. Each project will have its own characteristic flexibility thumbprint. In Chapter Nine, I explore this topic further and provide a means for actually balancing the needs for flexibility with the needs for structure.

Should you decide to use the Project Analyzer as portrayed in Figure 1.6 for your project, note that it was created for IT software projects. Although it seems relatively general, please modify it to the attributes of your project.

Summary

This chapter provides a foundation for flexible development. Some key points:

- I aim to provide customizable tools, techniques, and approaches that will help you accommodate—even embrace—change rather than suppressing or denying it.
- Change is essential to innovation, and industry's record over a recent fourteen-year period is one of decreasing product innovation.
- Not all projects or parts of a project need be flexible, and it can be inadvisable to make them all flexible. Use flexibility with discretion when its benefits outweigh its costs.
- Agile software development, Extreme Programming in particular, is a motivating model of the possibilities for increasing flexibility, but most agile practices do not translate directly to non-software products. This book develops similar practices for use outside the software industry.

―――――――

In contrast with this introductory chapter, the ones to follow present the tools and approaches of flexibility. Think of them as

a tool kit. Each chapter presents a category of tools. For a given project, you will need an assortment of tools but not necessarily all of them.

I present the categories of tools by chapter only because it is easier to assimilate them separately. Like the practices of XP though, they fit together and mutually support each other. They tend to be synergistic (1 + 1 = 3). Some may not seem to fit your business, but this may only be because you are not thinking creatively enough about them. For instance, at first, modular product architectures (Chapter Three) may not seem to apply to homogeneous chemical products like paint, but they may apply to the manufacturing or distribution processes of such products to make them more flexible.