2

Rapid-Development Strategy

Contents

Related Topics

Classic mistakes: Chapter 3
Development fundamentals: Chapter 4
Risk management: Chapter 5
Core issues in rapid development: Chapter 6

IF YOU TOOK 100 WORLD-CLASS MUSICIANS and put them into an orchestra without a conductor, they wouldn't sound like a world-class orchestra. The string section's timing wouldn't match that of the woodwind or brass section. Instructing the musicians to "do their best" wouldn't help them know whether they should play louder or softer. Such a musical event would be a waste of talent.

A similar waste of talent is commonplace in software development. Teams of smart, dedicated developers employ the latest best practices and still fail to meet their schedule goals.

One of the most tempting traps that people who want faster development fall into is the trap of focusing too much on a single schedule-oriented development practice. You might execute rapid prototyping perfectly, but if you make a mistake somewhere else—"Oops! We forgot to include time in the schedule for the printing subsystem!"—you will not achieve rapid development. Individual schedule-oriented practices are only part of what's needed to achieve the shortest possible schedule. A general framework that allows those practices to be used to maximum advantage is also needed.

This chapter puts forth an orchestrated strategy for achieving rapid development.

Case Study 2-1. *Rapid* Development Without a Clear Strategy

Mickey was ready to lead his second project at Square-Tech, a giant in the PC spreadsheet world. His previous project had been tough. The original schedule had called for his team to deliver Square-Calc version 2.0 in 12 months, but it had taken 18. The team had known at the outset that the date was aggressive, so for almost the entire 18 months they were on a death march, working 12-hour days and 6- or 7-day weeks. At the end of the project, two of the six team members quit and Bob, the strongest developer on the team, set out from Seattle on his bicycle for parts unknown. Bob said he wasn't quitting, and he sent Mickey a postcard from Ottumwa, South Dakota—a picture of himself riding a giant jackalope—but no one knew when he would be back.

Square-Calc 3.0 needed to be released 12 months after version 2.0, so after two months of project cleanup, post mortems, and vacations, Mickey was ready to try again. He had 10 months to deliver version 3.0. Mickey met with Kim, his manager, to discuss the project plan. Kim was known for being able to eke out every possible bit of work from the developers she managed. John, from user documentation, and Helen, from QA, were also present.

"Version 3.0 needs to leapfrog the competition," Kim said. "So we need a strong effort on this project. I know that your team didn't feel that the company was fully behind them last time, so this time the company is ready to provide all the support it can. I've approved individual private offices, new state-of-the-art computers, and free soda pop for the whole project. How does that sound?"

"That sounds great," Mickey said. "All these developers are experienced, so I mainly want to provide lots of motivation and support and then get out of the way. I don't want to micro-manage them. I'd like to have each developer sign up for a part of the system. We had a lot of interface problems last time, so I also want to spend some time designing the interfaces between the parts, and then turn them loose."

"If this is a 10-month project, we're going to need visually frozen software by the 8-month mark to get the user documentation ready on time," John said. "Last time the developers kept making changes right up until the end. The README file was 20 pages long, which was embarrassing. Our user manuals are getting killed in the reviews. As long as you agree to a visual freeze, your development approach sounds fine."

"We need visual freeze by about the same time to write our automated test scripts," Helen added. Mickey agreed to the visual freeze. Kim approved Mickey's overall approach and told him to keep her posted.

(continued)

Case Study 2-1. Rapid Development Without a Clear Strategy, *continued*

As the project began, the developers were happy about their private offices, new computers, and soda pop, so they got off to a strong start. It wasn't long before they were voluntarily working well into the evening.

Months went by, and they made steady progress. They produced an early prototype, and continued to produce a steady stream of code. Management kept the pressure on; John reminded Mickey several times of his commitment to a visual freeze at the 8-month mark, which Mickey found irritating, but everything seemed to be progressing nicely.

Bob returned from his bike trip during the project's fourth month, refreshed, and jumped into the project with some new thoughts he'd had while riding. Mickey worried about whether Bob could implement as much functionality as he wanted to in the time allowed, but Bob was committed to his ideas and guaranteed on-time delivery no matter how much work it took.

The team members worked independently on their parts, and as visual freeze approached, they began to integrate their code. They started at 2:00 in the afternoon the day before the visual-freeze deadline and soon discovered that the program wouldn't compile, much less run. The combined code had several dozen syntax errors, and it seemed like each one they fixed generated 10 more. At midnight, they decided to call it a night.

The next morning, Kim met with the team. "Is the program ready to hand over to documentation and testing?"

"Not yet," Mickey said. "We're having some integration problems. We might be ready by this afternoon." The team worked that afternoon and evening, but couldn't fix all of the bugs they were discovering. At the end of the day they conceded that they had no idea how much longer integration would take.

It took two full weeks to fix all the syntax errors and get the system to run at all. When the team turned over the frozen build two weeks late, testing and documentation rejected it immediately. "This is too unstable to document," John said. "It crashes every few minutes, and there are lots of paths we can't even exercise."

Helen agreed. "There's no point in having testers write defect reports when the system is so unstable that it crashes practically every time you make a menu selection."

Mickey agreed with them and said he'd focus his team's efforts on bug fixes. Kim reminded them of the 10-month deadline and said that this product couldn't be late like the last one.

It took a month to make the system reliable enough to begin documenting and testing it. By then they were only two weeks from the 10-month mark, and they worked even harder.

(continued)

Case Study 2-1. Rapid Development Without a Clear Strategy, *continued*

But testing began finding defects faster than the developers could correct them. Fixes to one part of the system frequently caused problems in other parts. There was no chance of making the 10-month ship date. Kim called an emergency meeting. "I can see that you're all working hard," she said, "but that's not good enough. I need results. I've given you every kind of support I know how, and I don't have any software to show for it. If you don't finish this product soon, the company could go under."

As the pressure mounted, morale faded fast. More months went by, the product began to stabilize, and Kim kept the pressure on. Some of the interfaces turned out to be extremely inefficient, and that called for several more weeks of performance work.

Bob, despite working virtually around the clock, delivered his software later than the rest of the team. His code was virtually bug-free, but he had changed some of the user-interface components, and testing and user documentation threw fits.

Mickey met with John and Helen. "You won't like it, but our options are as follows: We can keep Bob's code the way it is and rev the test scripts and user documentation, or we can throw out Bob's code and write it all again. Bob won't rewrite his code, and no one else on the team will either. Looks like you'll have to change the user documentation and test scripts." After putting up token resistance, John and Helen begrudgingly agreed.

In the end, it took the developers 15 months to complete the software. Because of the visual changes, the user documentation missed its slot in the printer's schedule, so after the developers cut the master disks there was a two-week shipping delay while Square-Tech waited for documents to come back from the printer. After release, user response to Square-Calc version 3.0 was lukewarm, and within months it slipped from second place in market share to fourth. Mickey realized that he had delivered his second project 50 percent over schedule, just like the first.

2.1 General Strategy for Rapid Development

The pattern described in Case Study 2-1 is common. Avoidance of the pattern takes effort but is within reach of anyone who is willing to throw out their bad habits. You can achieve rapid development by following a four-part strategy:

1. Avoid classic mistakes.
2. Apply development fundamentals.

3. Manage risks to avoid catastrophic setbacks.
4. Apply schedule-oriented practices such as the three kinds of practices shown in Figure 1-2 in Chapter 1.

As Figure 2-1 suggests, these four practices provide support for the best possible schedule.

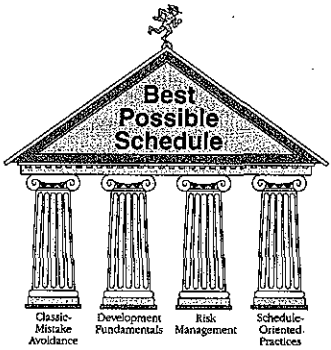

Figure 2-1. *The four pillars of rapid development. The best possible schedule depends on classic-mistake avoidance, development fundamentals, and risk management in addition to the use of schedule-oriented practices.*

Rapid product development is not a quick fix for getting one product—which is probably already late—to market [illegible] instead, it is a strategic capability that must be built from the ground up.
Preston G. Smith and Donald G. Reinertsen, Developing Products in Half the Time

Pictures with pillars have become kind of hokey, but the pillars in this picture illustrate several important points.

The optimum support for the best possible schedule is to have all four pillars in place and to make each of them as strong as possible. Without the support of the first three pillars, your ability to achieve the best possible schedule will be in jeopardy. You can use the strongest schedule-oriented practices, but if you make the classic mistake of shortchanging product quality early in the project, you'll waste time correcting defects when it's most expensive to do so. Your project will be late. If you skip the development fundamental of creating a good design before you begin coding, your program can fall apart when the product concept changes partway through development, and your project will be late. And if you don't manage risks, you can find out just before your release date that a key subcontractor is three months behind schedule. You'll be late again.

The illustration also suggests that the first three pillars provide most of the support needed for the best possible schedule. Not ideal support, perhaps, but most of what's needed. You might be able to achieve an optimal schedule without schedule-oriented practices.

Can you achieve the best possible schedule by focusing only on schedule-oriented practices? You might just be able to pull it off. People have pulled off stunts like that before. But as Figure 2-2 illustrates, it's a difficult balancing act. I can balance a chair on my chin, and my dog can balance a biscuit on his nose. But running a software project isn't a parlor trick, and if you rely on schedule-oriented practices to do all the work, you probably won't get all the support you need. If you do manage to pull it off once, you can't count on being able to do so again.

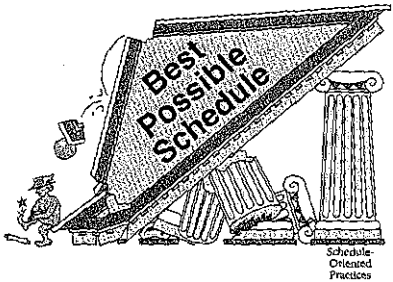

Figure 2-2. *Result of focusing solely on schedule-oriented practices. Even the finest schedule-oriented practices aren't strong enough to support the best possible schedule by themselves.*

The first three pillars shown in Figure 2-1 are critical to the success of rapid development, so I intend to lay out in very clear terms what I mean by classic-mistake avoidance, development fundamentals, and risk management. Chapter 3 will introduce classic mistakes. In most cases, simply being aware of a mistake will be enough to prevent it, so I will present a list of classic mistakes in that chapter. I'll take up the topic of development fundamentals in Chapter 4 and the topic of risk management in Chapter 5.

The rest of this book discusses specific schedule-oriented practices, including speed-oriented, schedule-risk-oriented, and visibility-oriented practices. The best-practice summaries in Part III of this book list the effect that each practice has on development speed, schedule risk, and visibility. If you'd rather read about rapid development itself before reading about the three steps needed to lay the groundwork for rapid development, you might skip ahead to Chapter 6, "Core Issues in Rapid Development," and to other chapters.

2.2 Four Dimensions of Development Speed

Whether you're bogged down trying to avoid mistakes or cruising at top speed with highly effective schedule-oriented practices, your software project operates along four important dimensions: people, process, product, and technology. People perform quickly, or they perform slowly. The process leverages people's time, or it throws up one stumbling block after another. The product is defined in such a way that it almost builds itself, or it is defined in a way that stymies the best efforts of the people who are building it. Technology assists the development effort, or it thwarts developers' best attempts.

You can leverage each of these four dimensions for maximum development speed. Figure 2-3 illustrates the point.

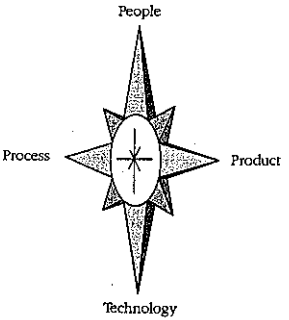

Figure 2-3. *The four dimensions of development speed—shown here in two dimensions. You can focus on all four dimensions at the same time.*

In response to this diagram, I can almost hear some engineers saying, "Hey! That's not four *dimensions*. It's four *directions*. You can't even draw in four dimensions!" Well, you're right. I can't draw in four dimensions, which is why I've shown this illustration in two dimensions. But the concept I want to get across is very much one of dimension rather than direction.

Software development books tend to emphasize one dimension and downplay the others, but there isn't necessarily a trade-off between emphases on people, process, product, and technology. If these were directions, a focus on people would detract from a focus on technology. A focus on product would detract from a focus on process. But because they're dimensions, you can focus on people, process, product, and technology all at the same time.

Software organizations tend to view the dimensions they don't focus on as fixed, and I think that's one reason that project planning can be so frustrating, especially schedule planning. When you focus on a single dimension, it can be nearly impossible to satisfy everyone's objectives. Truly rapid development requires you to incorporate a variety of kinds of practices (Boehm et al. 1984, Jones 1991). The organizations that are the most effective at achieving rapid development optimize all four rapid-development dimensions simultaneously.

Once you realize that each of the four dimensions can potentially provide tremendous leverage over a software schedule, your planning can become fuller, more creative, more effective, and better able to satisfy you and the other concerned parties.

The following subsections introduce the four dimensions and discuss the synergy among them.

People

Results of individual experiments on peopleware issues are well known. You might be familiar with the claim that there is at least a 10-to-1 difference in productivity among different developers. Or you might be familiar with the positive contribution that an explicit motivational improvement program can have.

What is less familiar to most developers, indeed to most people in the industry, is that the research on peopleware issues has been steadily accumulating over the past 15 to 20 years. It is now possible to step beyond the many individual-study conclusions and synthesize some general conclusions from trends in the research.

HARD DATA

The first conclusion is that we now know with certainty that peopleware issues have more impact on software productivity and software quality than any other factor. Since the late 1960s, study after study has found that the productivity of individual programmers with similar levels of experience does indeed vary by a factor of at least 10 to 1 (Sackman, Erikson, and Grant 1968, Curtis 1981, Mills 1983, DeMarco and Lister 1985, Curtis et al. 1986, Card 1987, Valett and McGarry 1989).

HARD DATA

Studies have also found variations in the performance of entire teams on the order of 3, 4, or 5 to 1 (Weinberg and Schulman 1974; Boehm 1981; Mills 1983; Boehm, Gray, and Seewaldt 1984). After 20 years of experimentation on live projects, researchers at NASA's Software Engineering Laboratory have concluded that technology is not the answer; the most effective practices are those that leverage the human potential of their developers (Basili et al. 1995).

Because it is so clear that peopleware issues strongly influence productivity, it is also now crystal clear that any organization that's serious about improving productivity should look first to the peopleware issues of motivation, teamwork, staff selection, and training. There are other ways to improve productivity, but peopleware offers the greatest potential benefit. If you are serious about rapid development, you have to be serious about peopleware issues. Taken collectively, peopleware issues matter more than process, product, or technology. You have to address them if you want to succeed.

This conclusion is a strong one, but it should not be taken as support for any peopleware initiative whatsoever. The research results simply say that the effects of individual ability, individual motivation, team ability, and team motivation dwarf other productivity factors. They do not say specifically that team T-shirts, free soda pop, windowed offices, productivity bonuses, or Friday afternoon beer busts improve motivation, but the implication is clear: any organization that wants to improve its productivity should be actively trying all these things.

This book deals with several ways that you can maximize human potential to reduce software schedules.

CROSS-REFERENCE For more on job matching and career progression, see "Work Itself" in Section 11.2. For more on team balance and problem personnel, see Chapter 12, "Teamwork," and Chapter 13, "Team Structure."

Staff selection for team projects. In his landmark book, *Software Engineering Economics*, Barry Boehm presents five principles of software staffing (Boehm 1981):

- *Top talent*—Use better and fewer people.
- *Job matching*—Fit the tasks to the skills and motivation of the people available.
- *Career progression*—Help people to self-actualize rather than forcing them to work where they have the most experience or where they are most needed.
- *Team balance*—Select people who will complement and harmonize with each other.
- *Misfit elimination*—Eliminate and replace problem team members as quickly as possible.

Other factors that can make a difference include people's design ability, programming ability, programming-language experience, machine and environment experience, and applications-area experience.

Team organization. The way that people are organized has a great effect on how efficiently they can work. Software shops can benefit from tailoring their teams to match project size, product attributes, and schedule goals. A specific software project can also benefit from appropriate specialization.

Motivation. A person who lacks motivation is unlikely to work hard and is more likely to coast. No factor other than motivation will cause a person to forsake evenings and weekends without being asked to do so. Few other factors can be applied to so many people on so many teams in so many organizations. Motivation is potentially the strongest ally you have on a rapid-development project.

Variations in Productivity

I refer to several ratios related to variations in productivity in this book, and keeping them straight can get confusing. Here's a summary of the variations that researchers have found:

- Greater than 20-to-1 differences in productivity among individuals with different depths and breadths of experience
- 10-to-1 differences in productivity among individuals with the same levels of experience
- 5-to-1 differences in productivity among groups with different levels of experience
- 2.5-to-1 differences in productivity among groups with similar levels of experience

Process

Process, as it applies to software development, includes both management and technical methodologies. The effect that process has on a development schedule is easier to assess than the effect that people have, and a great deal of work is being done by the Software Engineering Institute and other organizations to document and publicize effective software processes.

HARD DATA

Process represents an area of high leverage in improving your development speed—almost as much as people. Ten years ago it might have been reasonable to debate the value of a focus on process, but, as with peopleware, today the pile of evidence in favor of paying attention to process has become overwhelming. Organizations such as Hughes Aircraft, Lockheed, Motorola, NASA, Raytheon, and Xerox that have explicitly focused on improving their development processes have, over several years, cut their times-to-market by about one-half and have reduced cost and defects by factors of 3 to 10 (Pietrasanta 1991a, Myers 1992, Putnam and Myers 1992, Gibbs 1994, Putnam 1994, Basili et al. 1995, Raytheon 1995, Saiedian and Hamilton 1995).

Some people think that attention to process is stifling, and there's no doubt that some processes are overly rigid or overly bureaucratic. A few people have created process standards primarily to make themselves feel powerful. But that's an abuse of power—and the fact that a process focus can be abused should not be allowed to detract from the benefits a process focus

can offer. The most common form of process abuse is neglect, and the effect of that is that intelligent, conscientious developers find themselves working inefficiently and at cross-purposes when there's no need for them to work that way. A focus on process can help.

Rework avoidance. If requirements change in the late stages of project, you might have to redesign, recode, and retest. If you have design problems that you didn't find until system testing, you might have to throw away detailed design and code and then start over. One of the most straightforward ways to save time on a software project is to orient your process so that you avoid doing things twice.

HARD DATA

Raytheon won the IEEE Computer Society's Software Process Achievement Award in 1995 for reducing their rework costs from 41 percent to less than 10 percent and simultaneously tripling their productivity (Raytheon 1995). The relationship between those two feats is no coincidence.

CROSS-REFERENCE For more on quality assurance, see Section 4.3, "Quality-Assurance Fundamentals."

Quality assurance. Quality assurance has two main purposes. The first purpose is to assure that the product you release has an acceptable level of quality. Although that is an important purpose, it is outside the scope of this book. The second function of quality assurance is to detect errors at the stage when they are least time-consuming (and least costly) to correct. This nearly always means catching errors as close as possible to the time that they are introduced. The longer an error remains in the product, the more time-consuming (and more costly) it will be to remove. Quality assurance is thus an indispensable part of any serious rapid-development program.

CROSS-REFERENCE For more on development fundamentals, see Section 4.2, "Technical Fundamentals."

Development fundamentals. Much of the work that has been done in the software-engineering field during the last 20 years has been related to developing software rapidly. A lot of that work has focused on "productivity" rather than on rapid development per se, and, as such, some of it has been oriented toward getting the same work done with fewer people rather than getting a project done faster. You can, however, interpret the underlying principles from a rapid-development viewpoint. The lessons learned from 20 years of hard knocks can help your project to proceed smoothly. Although standard software-engineering practices for analysis, design, construction, integration, and testing won't produce lightning-fast schedules by themselves, they can prevent projects from spinning out of control. Half of the challenge of rapid development is avoiding disaster, and that is an area in which standard software-engineering principles excel.

CROSS-REFERENCE For more on risk management, see Chapter 5, "Risk Management."

Risk management. One of the specific practices that's focused on avoiding disaster is risk management. Developing rapidly isn't good enough if you get your feet knocked out from under you two weeks before you're scheduled to ship. Managing schedule-related risks is a necessary component of a rapid-development program.

Resource targeting. Resources can be focused effectively and contribute to overall productivity, or they can be misdirected and used ineffectively. On a rapid-development project, it is even more important than usual that you get the maximum bang for your schedule buck. Best practices such as productivity offices, timebox development, accurate scheduling, and voluntary overtime help to make sure that you get as much work done each day as possible.

CROSS-REFERENCE
For more on lifecycle planning, see Chapter 7, "Lifecycle Planning."

Lifecycle planning. One of the keys to targeting resources effectively is to apply them within a lifecycle framework that makes sense for your specific project. Without an overall lifecycle model, you can make decisions that are individually on target but collectively misdirected. A lifecycle model is useful because it describes a basic management plan. For example, if you have a risky project, a risk-oriented lifecycle model will suit you; and if you have vague requirements, an incremental lifecycle model may work best. Lifecycle models make it easy to identify and organize the many activities required by a software project so that you can do them with the utmost efficiency.

CROSS-REFERENCE
For more on customer orientation, see Chapter 10, "Customer-Oriented Development."

Customer orientation. One of the gestalt shifts between traditional, mainframe software development and more modern development styles has been the switch to a strong focus on customers' needs and desires. Developers have learned that developing software to specification is only half the job. The other half is helping the customer figure out what the product should be, and most of the time that requires an approach other than a traditional paper-specification approach. Putting yourself on the same side as the customer is one of the best ways to avoid the massive rework caused by the customer deciding that the product you just spent 12 months on is not the right product after all. The best practices of staged releases, evolutionary delivery, evolutionary prototyping, throwaway prototyping, and principled negotiation can all give you leverage in this area.

Who Is "The Customer"?

In this book, when I refer to "customers," I'm referring to the people who pay to have the software developed and the people who are responsible for accepting or rejecting the product. On some projects, those will be the same person or group; on others, they'll be different. On some projects, the customer is a real flesh-and-blood client who pays your project's development costs directly. On other projects, it's another internal group within your organization. On still other projects, the customer is the person who plunks down $200 for a shrink-wrap software package. In that case, the real customer is remote, and there is usually a manager or marketer who represents the customer to you.

Depending on your situation, you might understand the term "customer" to mean "client," "marketer," "end-user," or "boss."

Product

The most tangible dimension of the people/process/product/technology compass is the product dimension, and a focus on product size and product characteristics presents enormous opportunities for schedule reduction. If you can reduce a product's feature set, you can reduce the product's schedule. If the feature set is flexible, you might be able to use the 80/20 rule and develop the 80 percent of the product that takes only 20 percent of the time. You can develop the other 20 percent later. If you can keep the product's look and feel, performance characteristics, and quality characteristics flexible, you can assemble the product from preexisting components and write a minimum amount of custom code. The exact amount of schedule reduction made possible by focusing on product size and product characteristics is limited only by your customer's product concept and your team's creativity.

Both product size and product characteristics offer opportunities to cut development time.

CROSS-REFERENCE For more on manipulating product size to support development speed, see Chapter 14, "Feature-Set Control." For more on the effect that product size has on a development schedule, see Chapter 8, "Estimation."

Product size. Product size is the largest single contributor to a development schedule. Large products take a long time. Smaller products take less time. Additional features require additional specification, design, construction, testing, and integration. They require additional coordination with other features, and they require that you coordinate other features with them. Because the effort required to build software increases disproportionately faster than the size of the software, a reduction in size will improve development speed disproportionately. Cutting the size of a medium-size program by one-half will typically cut the effort required by almost *two-thirds*.

You can reduce product size outright by striving to develop only the most essential features, or you can reduce it temporarily by developing a product in stages. You can also reduce it by developing in a higher-level language or tool set so that each feature requires less code.

CROSS-REFERENCE For more on the effect that goals can have on a development schedule, see "Goal Setting" in Section 11.2.

Product characteristics. Although not as influential as product size, other product characteristics do have an effect on software schedules. A product with ambitious goals for performance, memory use, robustness, and reliability will take longer to develop than a product without any goals for those characteristics. Choose your battles. If rapid development is truly top priority, don't shackle your developers by insisting on too many priorities at once.

Technology

CROSS-REFERENCE For more on productivity tools, see Chapter 15, "Productivity Tools."

Changing from less effective tools to more effective tools can also be a fast way to improve your development speed. The change from low-level languages like assembler to high-level languages like C and Pascal was one of the most influential changes in software-development history. The current move toward componentware (VBXs and OCXs) might eventually produce

similarly dramatic results. Choosing tools effectively and managing the risks involved are key aspects of a rapid-development initiative.

Synergy


HARD DATA

There is a point at which your focus on people, process, product, and technology becomes synergistic. Neil Olsen conducted a study in which he found that going from low spending to medium spending on staffing, training, and work environment produced proportionate gains: additional spending was justified on roughly a 1-to-1 payback basis. But when spending on staffing, training, and work environment went from medium to high, productivity skyrocketed, paying back 2 to 1 or 3 to 1 (Olsen 1995).

Software-engineering practices can also be synergistic. For example, an organizationwide coding standard helps an individual project, but it also makes it easier for one project to reuse components from another project. At the same time, a reusable-components group can help to enforce a coding standard and ensure that it's meaningful across projects. Design and code reviews help to disseminate knowledge about both the coding standard and existing reusable components, and they promote the level of quality needed for reuse to succeed. Good practices tend to support one another.

2.3 General Kinds of Fast Development

Different situations call for different levels of commitment to development speed. In some cases, you'd like to increase development speed if you can do it easily and without additional cost or product degradation. In other cases, circumstances call for you to increase development speed at all costs. Table 2-1 describes some trade-offs among different development approaches.

Table 2-1. Characteristics of Standard Approaches to Schedule-Oriented Development

	Effect of Development Approach On...		
Development Approach	**...Schedule**	**...Cost**	**...Product**
Average practice	Average	Average	Average
Efficient development (balancing cost, schedule, and functionality)	Better than average	Better than average	Better than average
Efficient development (tilted toward best schedule)	Much better than average	Somewhat better than average	Somewhat better than average
All-out rapid development	Fastest possible	Worse than average	Worse than average

Efficient Development

CROSS-REFERENCE
For an example of the benefits of efficient development, see Section 4.2, "Technical Fundamentals" and Chapter 4, "Software Development Fundamentals," generally.

As you can see from Table 2-1, average practice is...average. The second approach listed in the table is what I call "efficient development," which is the combination of the first three pillars of maximum development speed as shown in Figure 2-4. That approach produces better than average results in each of the three categories. Many people achieve their schedule goals after they put the first three pillars into place. Some people discover that they didn't need rapid development after all; they just needed to get organized! For many projects, efficient development represents a sensible optimization of cost, schedule, and product characteristics.

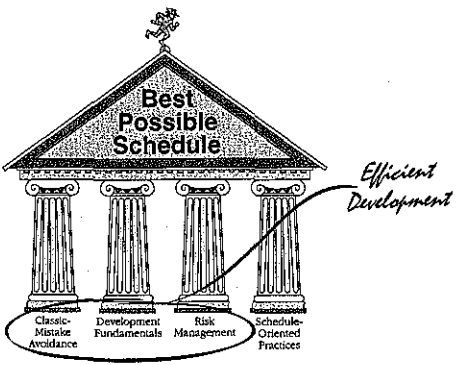

Figure 2-4. *Efficient development. The first three steps in achieving the best possible schedule make up "efficient development." Many project teams find that efficient development provides all the development speed they need.*

Can you achieve shorter schedules without first attaining efficient development? Maybe. You can choose effective, schedule-oriented practices and avoid slow or ineffective practices without focusing on efficient development per se. Until you attain efficient development, however, your chances of success in using schedule-oriented practices will be uncertain. If you choose specific schedule-oriented practices without a general strategy, you'll have a harder time improving your overall development capability. Of course, only you can know whether it's more important to improve your overall development capabilities or to try completing a specific project faster.

CROSS-REFERENCE
For more on the relationship between quality and development speed, see Section 4.3, "Quality-Assurance Fundamentals."

Another reason to focus on efficient development is that for most organizations the paths to efficient development and shorter schedules are the same. For that matter, until you get to a certain point, the paths to shorter schedules, lower defects, and lower cost are all the same, too. As Figure 2-5 shows, once you get to efficient development the roads begin to diverge, but from where they are now, most development groups would benefit by setting a course for efficient development first.

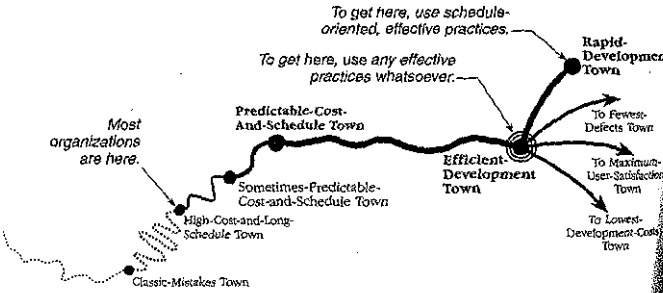

Figure 2-5. *The road to rapid development. From where most organizations are now, the route to rapid development follows the same road as the route to fewest defects, maximum user satisfaction, and lowest development costs. After you reach efficient development, the routes begin to diverge.*

Efficient Development Tilted Toward Best Schedule

CROSS-REFERENCE
For more on deciding between speed-oriented and schedule-risk–oriented practices, see Section 1.2, "Attaining Rapid Development," and Section 6.2, "What Kind of Rapid Development Do You Need?"

The third development approach listed in Table 2-1 is a variation of efficient development. If you are practicing efficient development and find that you still need better schedule performance, you can choose development practices that are tilted toward increasing development speed, reducing schedule risk, or improving progress visibility. You'll have to make small trade-offs in cost and product characteristics to gain that speed or predictability; if you start from a base of efficient development, however, you'll still be much better off than average.

All-Out Rapid Development

CROSS-REFERENCE
For more on nominal schedules, see "Nominal Schedules" in Section 8.6. For more on the costs of schedule compression, see "Two Facts of Life" in Section 8.6.

The final schedule-oriented development approach is what I call "all-out rapid development"—the combination of efficient and *inefficient* schedule-oriented practices. There comes a point when you're working as smart as you can and as hard as you can, and the only thing left to do at that point is to pay more, reduce the feature set, or reduce the product's polish.

Here's an example of what I mean by an "inefficient" practice: you can compress a project's nominal development schedule by 25 percent simply by adding more people to it. Because of increased communications and management overhead, however, you have to increase your team size by about 75 percent to achieve that 25-percent schedule reduction. The net effect of a shorter schedule and larger team size is a project that costs 33 percent more than the nominal project.

CROSS-REFERENCE For more on whether you need all-out rapid development, see Section 6.2, "What Kind of Rapid Development Do You Need?"

The move to all-out rapid development is a big step and requires that you accept increased schedule risk or large trade-offs between cost and product characteristic—or both. Few projects welcome such trade-offs, and most projects are better off just choosing some form of efficient development.

2.4 Which Dimension Matters the Most?

CROSS-REFERENCE For more on customizing software processes to the needs of specific projects, see Section 6.1, "Does One Size Fit All?"

Boeing, Microsoft, NASA, Raytheon, and other companies have all learned how to develop software in ways that meet their needs. At the strategy level, these different organizations have a lot in common. They have learned how to avoid classic mistakes. They apply development fundamentals. And they practice active risk management. At the tactical level, there is a world of difference in the ways that each of these successful organizations emphasize people, process, product, and technology.

Different projects have different needs, but the key in all cases is to accept the limitations on the dimensions you can't change and then to emphasize the other dimensions to get the rest of the schedule benefit you need.

If you're developing a fuel-injection system for a car, you can't use 4GLs or a visual programming environment to develop the real-time, embedded software; you need greater performance and better low-level control than these tools can provide. You're prevented from exercising the technology dimension to the utmost. Instead, you have to emphasize technology as much as you can—and then get your real leverage from the people, process, and product dimensions.

If you're working on an in-house business program, perhaps you can use a 4GL, a visual programming environment, or a CASE tool. You're able to exercise technology to the utmost. But you might work for a stodgy corporation that prevents you from doing much in the people dimension. Emphasize people as much as the company allows, and then get the remaining leverage you need from the product and process dimensions.

If you're working in a feature-driven shrink-wrap market, you might not be able to shrink your feature set much to meet a tight schedule. Shrink it as much as you can, and then emphasize people, process, and technology to give you the rest of what you need to meet your schedule.

Kinds of Projects—Systems, Business, and Shrink-Wrap

This book describes three general kinds of projects, *systems*, *business*, and *shrink-wrap*.

Systems software includes operating system software, device drivers, compilers, and code libraries. For purposes of this book (and despite their differences), *embedded software*, *firmware*, *real-time systems*, and *scientific software* share many characteristics with systems software.

Business software refers to in-house systems that are used by a single organization. They run on a limited set of hardware, perhaps only a single computer. Payroll systems, accounting systems, and inventory control systems are typical examples. In this book, I treat *IS*, *IT*, and *MIS* software as belonging to the general category of business software.

Shrink-wrap software is software that is packaged and sold commercially. It includes both horizontal-market products like word processors and spreadsheets and vertical-market products like financial-analysis, screenplay-writing, and legal case-management programs.

I use a few other terms to refer to kinds of software that aren't described by these three general labels. *Commercial* software is any kind of software developed for commercial sale. *In-house* software is software that is developed solely for in-house use and is not for commercial sale. *Military* software is written for use by the military. *Interactive* software is any software with which a user can interact directly, which includes most of the software being written today.

To summarize: analyze your project to determine which of the four dimensions are limited and which you can leverage to maximum advantage. Then stretch each to the utmost. That, in a nutshell, is the key to successful rapid development.

2.5 An Alternative Rapid-Development Strategy

CROSS-REFERENCE
For more on commitment-based approaches, see "Commitment-Based Scheduling" in Section 8.5 and Chapter 34, "Signing Up."

The approach to rapid development that I lay out in this book is not the only approach that has ever been known to work. There is a different road that some projects have traveled successfully. That road is characterized by hiring the best possible people, asking for a total commitment to the project, granting them nearly total autonomy, motivating them to an extreme degree, and then seeing that they work 60, 80, or even 100 hours a week until either they or the project are finished. Rapid development with this commitment-based approach is achieved through grit, sweat, and determination.

This approach has produced notable successes, including Microsoft NT 3.0 and the Data General Eagle Computer, and it is unquestionably the most popular approach to rapid development in use today. For a start-up company

with cash-flow concerns, it has the advantage of extracting two months' work from an employee for one month's pay. That can mean the difference between finishing a killer product in time to meet a marketing window that allows the company to make a fortune and running out of money before the team even finishes the product. By keeping team size small, it also reduces communications, coordination, and management overhead. If practiced with a keen awareness of the risks involved and with some restraint, the approach can be successful.

Unfortunately, this approach is hardly ever practiced that carefully. It usually devolves to a code-like-hell approach, which produces drawn-out projects that seem to last forever. The approach is a quick fix, and it shares many problems with other quick fixes. I criticize aspects of the approach throughout the book. Here's a summary.

The approach is hit-or-miss. Sometimes it works; sometimes it doesn't. The factors that make it work or not work are largely impossible to control. When you do manage to complete a project, sometimes you get the functionality you planned for; sometimes you're surprised. Sometimes you hit your quality targets; sometimes you don't. The approach makes specific product characteristics difficult to control.

It causes long-term motivation problems. On commitment-based projects, developers start out enthusiastically and put in ever-increasing overtime as they begin to fully realize what it will take to meet their commitments. Eventually, long hours aren't enough, and they fail to meet their schedule commitments. Their morale fades as, one by one, they are forced to admit defeat.

Once developers have put their hearts and souls into trying to meet their commitments and have failed, they become reluctant to make additional strong commitments. They start to resent the overtime. They make additional commitments with their mouths but not with their hearts, and the project loses any semblance of planning or control. It is not uncommon for a project that has reached this state to stay "three weeks from completion" for six months or more.

It's unrepeatable. Even if the code-like-hell approach succeeds once, it doesn't lay the groundwork for succeeding next time. Because it burns people out, it more likely lays the groundwork for future failures. A company cannot easily repair the human damage inflicted by such a project, and accounts of such projects invariably report that massive staff turnover accompanies them (see, for example, Kidder 1981, Carroll 1990, Zachary 1994).

It's hard on nonsoftware organizations. Because it's based on individual heroics rather than on coordination, cooperation, and planning, the code-like-hell approach provides little visibility or control to other stakeholders in the project. Even when you succeed in developing the software faster than

average, you have no way of knowing how long the project will take. You don't know when you'll be done until you're actually done.

Some of the speed benefit arising from the commitment-based approach is neutralized because other groups that must coordinate with software developers—including testing, user documentation, and marketing—can't plan. In a code-like-hell project, frustration about the inability to get reliable schedule information from developers causes tempers to flare, and people within the development team are set against people outside the team. What's good for the software part of the project isn't necessarily good for the project overall.

It wastes human resources extravagantly. Developers who participate in this kind of project forego families, friends, hobbies, and even their own health to make a project successful. Severe personality conflicts are the rule rather than the exception. This level of sacrifice might be justifiable to win a war or put a man on the moon, but it isn't needed to develop business software. With few exceptions, the sacrifices aren't necessary: the same results can be achieved through careful, thoughtful, knowledgeable management and technical planning—with much less effort.

Table 2-2 summarizes some of the differences between the code-like-hell approach and the approach this book describes.

Table 2-2. Code-Like-Hell Approach Compared to This Book's Approach

Code-Like-Hell	This Book's Approach
Proponents claim incredible, instant improvement in development time.	Proponents claim modest instant improvement followed by greater, long-term improvement.
Requires little technical expertise beyond coding knowledge.	Requires significant technical expertise beyond coding knowledge.
High risk: frequently fails even when done as effectively as possible.	Low risk: seldom fails when done effectively.
Others will see you as "radical, dude." You'll look like you're giving your all.	Others will see you as conservative, boring, even old-fashioned. You won't look like you're working as hard.
Wastes human resources extravagantly.	Uses human resources efficiently and humanely.
Provides little progress visibility or control. You know you're done when you're done.	Permits tailoring the approach to provide as much visibility and control as you want.
Approach is as old as software itself.	Key parts of the approach used successfully for 15 years or more

Source: Inspired by "Rewards of Taking the Path Less Traveled" (Davis 1994).

The approach this book describes achieves rapid development through careful planning, efficient use of available time, and the application of schedule-oriented development practices. Overtime is not uncommon with this kind of rapid development, but it does not begin to approach the mountain of overtime commonly found when the code-like-hell approach is used. Efficient development commonly produces a shorter-than-average schedule. When it fails, it fails because people lose their resolve to continue using it, not because the practice itself fails.

In short, the code-like-hell approach guarantees extraordinary sacrifice but not extraordinary results. If the goal is maximum development speed rather than maximum after-hours hustle, any betting person has to favor efficient development.

If you read between the lines in this book, you'll find all the information you need to conduct a code-like-hell project as successfully as possible. A person could certainly transform this book's Dr. Jekyll into code-like-hell's Mr. Hyde. But I personally have had more than my fill of that kind of development, and I'm not going to spell out how to do it!

Case Study 2-2. Rapid Development with a Clear Strategy

Across the company from the Square-Calc 3.0 project, Sarah was ramping up the Square-Plan 2.0 project. Square-Plan was Square-Tech's project-management package. Sarah was the technical lead.

At the first team meeting, she introduced the team members and got right down to business. "I've gone all over the company collecting project postmortems," she said. "I've got a list a mile long of all the mistakes that other projects in this company have made. I'm posting the list here in the conference room, and I'd like you to raise a flag if we start to make any of them. I'd also like you to add any other potential mistakes you already know about or learn about as we go along. There's no point in repeating history if we don't have to.

"I selected you for this team because each one of you is strong in development fundamentals. You know what it means to do a good job of requirements gathering and design so that we won't waste time on needless rework downstream. I want everybody on this project to work smart instead of working hard. People who work too hard make too many mistakes, and we don't have time for that.

"I've also put together a risk-management plan. We've got an aggressive schedule, so we can't afford to be caught off guard by risks we could have prevented. The top risk on this list is that the schedule might be unachievable. I want us to reassess the schedule at the end of the week, and if it's unachievable, we'll come up with something more realistic."

(continued)

Case Study 2-2. Rapid Development with a Clear Strategy, *continued*

Everybody on the team nodded. To the people who had been through death-march projects, Sarah's talk felt like a breath of fresh air.

Later that week, Sarah met with her boss, Eddie. "The team has taken a hard look at the project's schedule, Eddie, and we've concluded that we have only about a five-percent chance of making our current deadline with the current feature set. That's assuming nothing changes, and of course a few things always change."

"That's no good," Eddie said. Eddie had a reputation for delivering what he promised. "I want at least a 50/50 chance of delivering the software on time. And I want to be able to respond to changes in the marketplace over the next 12 months. What do you recommend?"

"We haven't completely specified the product yet, so there's some flexibility there," Sarah said. "But we think the current set of requirements will take 10 to 30 months. I know that's a broad range, but that's the best we can do before we know more about what we're building. We need to have a product in 12 months, right? Considering that, I think we should add another developer and then set up an evolutionary delivery plan where we plan to build a shippable version of the software every two months, with our first delivery at the 8-month mark."

"That sounds good to me," Eddie said. "Besides, I think that functionality might be more important than schedule on this project. Let me talk with some people and get back to you."

When Eddie got back to Sarah, he told her that the company was willing to stretch the software schedule to 14 months to get the features it wanted but that she should still use the evolutionary-delivery plan to be safe. Sarah was relieved and said that she thought that was a more realistic target.

Over the first few weeks of the project, her team built a detailed, Hollywood-facade user-interface prototype. The "mistakes list" warned that sometimes a prototyping effort could take on a life of its own, so they set rigid timeboxes for prototyping work to avoid gold-plating the prototype. They used the prototype to interview potential customers about the candidate features and revised the prototype several times in response to user feedback.

Sarah continued to maintain the risks list and determined that the three key risks to the project were low quality that would cause excessive rework and lengthen the schedule, aggressiveness of the schedule, and features added by the competition between now and the end of the project. Sarah felt that the quality risk was addressed by the evolutionary-delivery plan. They would hand the first version of their software to QA at the 8-month mark, and QA could evolve their test cases along with the software.

The team addressed the schedule risk by creating a prioritized list of features. They would develop as much as they could in 14 months, but by bringing the

(continued)

Case Study 2-2. Rapid Development with a Clear Strategy, *continued*

product to a shippable state every two months, they would be guaranteed to have something to ship when they were supposed to. They also made design decisions for several features that were specifically intended to save implementation time. The less time-consuming implementations of the features would not be as slick, but they would be acceptable, and they made for a significant reduction in schedule risk.

The team addressed the competitive-features risk in two ways. They spent about five months developing a design that included a framework capable of supporting all the features they'd prototyped and a few more features that they thought they would include in version 2.0. Their design was intended to accommodate changes with little difficulty. They also allocated time at the 12-month mark to review competitors' products, revise the prototype, and implement competitively necessary features in the final two months.

At the 6-month mark, with the design complete, the team mapped out a set of miniature milestones that marked a path they would follow to release their first shippable version to testing at the 8-month mark. The 8-month version didn't do much, but its quality was good, and it provided a good foundation for further work. When that was behind them, the team mapped out another set of miniature milestones to get to the 10-month mark. The team used the same approach to get to the 12-month mark.

At the 12-month mark, the team reviewed competing products as planned. A competitor had released a good product at the 10-month mark, and it contained some features that Square-Plan 2.0 needed to include to be competitive. The team added the new features to their prioritized list, reprioritized, and mapped out miniature milestones for the final two months.

At about the same time, José, one of the junior developers, discovered a slightly better organization of one of the product's dialog boxes and brought the issue up at a staff meeting. George, one of the more senior developers, responded, "Your idea is great. I think we should change it, but we can't change it now, José. It's a 1-day change for you, but it would affect the documentation schedule by a week or more. How about putting it on the list for version 3.0?"

"I hadn't thought about the effect on the documentation schedule," José said. "That's a good point. I'll just submit it as a to-be-done-later change request."

At the 14-month mark, the team delivered its final software as planned. Square-Plan's quality was excellent because it had been in testing since Month 8. Documentation had been able to base its efforts on the detailed user-interface prototype while they waited for the live software, and the documentation was ready at the same time as the software. The developers had not had time to implement several low-priority features, but they had implemented all of the important ones. Square-Plan 2.0 was a success.

Further Reading

I know of no general books that discuss the topics of product or technology as they have been described in this chapter. This book discusses the topics further in Chapter 14, "Feature-Set Control," Chapter 15, "Productivity Tools," and Chapter 31, "Rapid-Development Languages."

The next three books provide general information on peopleware approaches to software development. The first is the classic.

DeMarco, Tom, and Timothy Lister. *Peopleware: Productive Projects and Teams.* New York: Dorset House, 1987.

Constantine, Larry L. *Constantine on Peopleware.* Englewood Cliffs, N.J.: Yourdon Press, 1995.

Plauger, P. J. *Programming on Purpose II: Essays on Software People.* Englewood Cliffs, N.J.: PTR Prentice Hall, 1993.

The following books provide information on software processes, the first at an organizational level, the second at a team level, and the third at an individual level.

Carnegie Mellon University/Software Engineering Institute. *The Capability Maturity Model: Guidelines for Improving the Software Process.* Reading, Mass.: Addison-Wesley, 1995. This book is a summary of the Software Engineering Institute's latest work on software process improvement. It fully describes the five-level process maturity model, benefits of attaining each level, and practices that characterize each level. It also contains a detailed case study of an organization that has achieved the highest levels of maturity, quality, and productivity.

Maguire, Steve. *Debugging the Development Process.* Redmond, Wash.: Microsoft Press, 1994. Maguire's book presents a set of folksy maxims that project leads can use to keep their teams productive, and it provides interesting glimpses inside some of Microsoft's projects.

Humphrey, Watts S. *A Discipline for Software Engineering.* Reading, Mass.: Addison-Wesley, 1995. Humphrey lays out a personal software process that you can adopt at an individual level regardless of whether your organization supports process improvement.