31422 01 pp. 001-010 r2ah.ps 2/9/05 5:21 PM Page 1

1 The Personal Process Strategy

“This is an absolutely critical project and it must be completed in nine months.” The division general manager was concluding his opening remarks at the kickoff for a project launch. As the Software Engineering Institute (SEI) coach, I had asked him to start the meeting by explaining why the business needed this product and how important it was to the company. As he told the development team, the company faced serious competition, and marketing could not hold the line for more than nine months. He then thanked me and the SEI for helping them launch this project and turned the meeting over to me.

After describing what the team would do during the next four days, I asked the marketing manager to speak. He described the new features needed for this product and how they compared with the current product and the leading competitor’s new product. He explained that this new competitive machine had just hit the market and was attracting a lot of customer interest. He said that if this team did not develop a better product within nine months, they would lose all of their major customers. He didn’t know how to hold the line.

On that happy note, we ended the meeting. The nine developers, the team leader, and I went to another room where we were to work. The team was scheduled to present its project plan to management in four days.

The developers were very upset. They had just finished one “death march” project and didn’t look forward to another. After they calmed down a bit, Greg, the team leader, asked me, “Why make a plan? After all, we know the deadline is in nine months. Why waste the next four days planning when we could start to work right now?”

“Do you think you can finish in nine months?” I asked.

“Not a chance,” he said.

“Well, how long will it take?”

“I don’t know,” he answered, “but probably about two years—like the last project.”

So I asked him, “Who owns the nine-month schedule now?”

“The general manager,” he said.

“OK,” I said, “Now suppose you go back and tell him that nine months is way too little time. The last project took two years and this is an even bigger job. What will happen?”

“Well,” Greg said, “he will insist that nine months is firm. Then I will tell him that we will do our very best.”

“Right,” I said, “that’s what normally happens. Now who would own the nine-month schedule?”

“Oh,” he answered, “I would.”

“Suppose you promise a nine-month schedule and then take two or more years. Would that cause problems?”

“It sure would,” Greg said. “Nobody would believe us again and our best customers would almost certainly jump ship.”

“Listen,” I explained, “management doesn’t know how long this project will take, and neither do you. As long as you are both guessing, the manager will always win. The only way out of this bind is for you to make a plan and do your utmost to accomplish what management wants. Then, if you can’t meet his deadline, you will understand why and be able to defend your schedule.”

Then Ahmed, one of the development team members, jumped in. “OK, so we need a plan, but how can we make one if we don’t even know the requirements?”

“Very good question,” I answered, “but do you have to commit to a date?”

“Well, yes.”

“So,” I asked, “you know enough to commit to a date but not enough to make a plan?”

He agreed that this didn’t make much sense. If they knew enough to make a commitment, they certainly ought to know enough to make a plan. “OK,” he said, “but without the requirements, the plan won’t be very accurate.”

“So,” I asked, “when could you make the most accurate plan?”

“Well, I suppose that would be after we have done most of the work, right?”

“That’s right,” I replied, “and that’s when you least need a plan. Right now, your plan will be least accurate but now is when you most need a schedule that you can defend. To do this, you must have a detailed plan.” With that, the team agreed to make a plan.

This was one of our early Team Software Process (TSP) teams and the members had all been trained in the Personal Software Process (PSP). They then used the PSP methods described in this book to make a plan that they believed was workable, and they then negotiated that plan with management. Throughout the project, they continued to use the PSP methods and, in the end, they delivered a high-quality product six weeks ahead of the 18-month schedule management had reluctantly accepted.

While this book is about the PSP, most PSP-trained software engineers end up working on TSP teams. This chapter describes how the PSP develops the personal disciplines you will later need. Then, after you are familiar with the PSP methods, Chapter 14 explains how you will apply these disciplines when you work on a TSP team. It also shows some of the results that TSP teams have achieved and describes how the PSP helped them accomplish such impressive work.

1.1 The PSP's Purpose

The PSP is a self-improvement process that helps you to control, manage, and improve the way you work. It is a structured framework of forms, guidelines, and procedures for developing software. Properly used, the PSP provides the data you need to make and meet commitments, and it makes the routine elements of your job more predictable and efficient.

The PSP's sole purpose is to help you improve your software engineering skills. It is a powerful tool that you can use in many ways. For example, it will help you manage your work, assess your talents, and build your skills. It can help you to make better plans, to precisely track your performance, and to measure the quality of your products. Whether you design programs, develop requirements, write documentation, or maintain existing software, the PSP can help you to do better work.

Rather than using one approach for every job, you need an array of tools and methods and the practiced skills to use them properly. The PSP provides the data and analysis techniques you need to determine which technologies and methods work best for you.

The PSP also provides a framework for understanding why you make errors and how best to find, fix, and prevent them. You can determine the quality of your reviews, the defect types you typically miss, and the quality methods that are most effective for you.

After you have practiced the exercises in this book, you will be able to decide what methods to use and when to use them. You will also know how to define, measure, and analyze your own process. Then, as you gain experience, you can enhance your process to take advantage of any newly developed tools and methods.

The PSP is not a magical answer to all of your software engineering problems, but it can help you identify where and how you can improve. However, you must make the improvements yourself.

1.2 The Logic for a Software Engineering Discipline

Software has become a critical part of many of the systems on which modern society depends. Everyone seems to need more and better software faster and cheaper. Many development projects are now so large and complex that a few brilliant specialists can no longer handle them. Unfortunately, there is no sign of a magical new technology to solve these problems. We must improve the quality and predictability of our work or society will have to either forgo these more sophisticated systems or suffer the damages caused by unsafe, unreliable, and insecure software-intensive systems.

The intuitive software development methods generally used today are acceptable only because there are no alternatives. Most software professionals are outstandingly creative but a few do really poor work. And, not surprisingly, poor practices produce poor products. Most software products can be made to work, but only after extensive testing and repair. From a scientific viewpoint, the process is distressingly unpredictable. It is much like the Brownian motion of particles in a gas. Here, physicists cannot predict what any individual particle will do, but they can statistically characterize the behavior of an entire volume of particles. This analogy suggests that large-scale software development should be treated as a problem of crowd control: don’t worry about what each individual does as long as the crowd behaves predictably.

This approach, while generally tolerated, has been expensive. An intuitive software process leaves the quality of each individual’s work a matter of blind luck. Professionals generally develop their own private methods and techniques. There are no disciplined frameworks, no sets of acceptable standards, no coaching systems, and no conducted rehearsals. Even agreement on what would characterize “good” professional performance is lacking. Software developers are left to figure out their own working methods and standards without the guidance and support that professionals find essential in sports, the performing arts, and medicine.

This situation becomes critical when each individual’s contribution is uniquely important. A symphony orchestra best illustrates this idea. Although the orchestra’s overall performance is a careful blend of many instruments, each musician is a highly competent and disciplined contributor. Individual performers occasionally stand out, but their combined effect is far more than the sum of these parts. What is more, any single sour note by any individual could damage the entire performance.

Unlike the musician, the software engineer must be part composer as well as performer. Like an orchestral performance, however, the performance of a software system can be damaged by almost any defective part. Because computers today possess extraordinary computational power, one badly handled interrupt or pointer could eventually cause an entire system to crash.

As our products become larger and more complex and as they are used for increasingly critical applications, the potential for damaging errors increases. The software industry has responded to this threat with increasingly rigorous and time-consuming tests. However, this testing strategy has not produced either safe or secure products. New large-scale software systems commonly have many security vulnerabilities. The suppliers have reacted by quickly producing and distributing fixes for the identified problems. Although this strategy has worked in the past, the hacker community has learned to launch attacks between the fix notice and user installation. In addition, because the fixes are themselves often defective and because it is very expensive to make systemwide updates, many organizations cannot afford to keep their systems current.

The only practical answer is to produce higher-quality products. Because the current test-and-fix strategy is incapable of producing products of suitable quality, this is now a problem for the software profession. The only responsible solution is to improve the working disciplines of each software professional.

In most professions, competency requires demonstrated proficiency with established methods. It is not a question of creativity versus skill. In many fields, creative work simply is not possible until one has mastered the basic techniques. Well-founded disciplines encapsulate years of knowledge and experience. Beginning professionals in the performing arts, high-energy physics, and brain surgery, for example, must demonstrate proficiency with many techniques before they are allowed to perform even the most routine procedures. Flawless skill, once acquired, enhances creativity. A skilled professional in such fields can outperform even the most brilliant but untrained layperson.

The PSP strategy is to improve the performance of practicing software engineers. A disciplined software engineering organization has well-defined practices. Its professionals use those practices, monitor and strive to improve their personal performance, and hold themselves responsible for the quality of the products they produce. And most important, they have the data and self-confidence required to resist unreasonable commitment demands.

Practiced disciplines have the further benefit of making software engineering more fun. Developing programs can be highly rewarding. Getting some clever routine to work is an achievement, and it is enormously satisfying to see a sophisticated program do what you intended. This satisfaction, however, is often diluted by the treadmill of debugging and the constant embarrassment of missed commitments. It is not fun to repeatedly make the same mistakes or to produce poor-quality results. Regardless of how hard you worked, nobody appreciates a late, over-budget, or poorly performing product.

31422 01 pp. 001-010 r2ah.ps 2/9/05 5:21 PM Page 6

Although developers are initially nervous about the structure and discipline of the PSP, they soon find it helpful. They quickly learn to make better plans and to achieve them. They find that the PSP discipline accelerates routine planning and project management while giving them more time for requirements and design. They also find that they can do things they never thought possible. Their reaction is like one developer who wrote on his PSP exercise, "Wow, I never did this before!" He had just written a program containing several hundred lines of code that compiled and tested without a single defect. The PSP can help to make software engineering the fun it should be.

1.3 Using Disciplined Development Practices

To consistently follow disciplined software engineering practices, we need the guidance of a defined and measured process that fits our project's needs and is convenient and easy to use. We also need the motivation to consistently use this process in our personal work.

A defined process specifies precisely how to do something. If we have not done such work before and we don't have a defined process readily available, we will not know the proper way to proceed. For example, if management gives us an aggressive deadline and we want to do the job in the best possible way, we need to understand the basic requirements before starting on the design or coding work. Unfortunately, without prior experience or a lot of guidance, few of us would know precisely how to do the requirements work. It would probably take us several days or weeks to figure it out.

While we are learning how to do this requirements work, the project clock is ticking and we have made no headway with the requirements, design, or implementation. Therefore, when faced with a tight schedule, what can we do? Like most developers, we would do what we know how to do: design and code. We would also hope that somebody else would clear up the requirements problems before we got too far along with development. Unfortunately, that rarely happens and we often end up building the wrong product. That is one reason why nearly 25% of all software projects are cancelled before development is completed.

1.4 Operational Processes

When people talk about processes, they usually talk in general terms about things such as prototyping, the spiral model, or configuration management. We all know what they mean in general terms, but to actually use one of these processes we need to know precisely what to do and the order in which to do it. To do this, we must have what is called an **operational process** (Deming 1982).

Most developers object to the idea of using an operational process. They think of a process as something that is imposed on them. We all object to being told how to do our job, particularly by someone who doesn't know as much about the work as we do. Although following an operational process sounds fine in theory, the real question is "Whose process is it?" That is what this book is all about—learning how to define and follow the operational processes you select to do your work. That is what turns an operational process into a convenient and useful personal process. In fact, it is the process that you *want* to use to do your *own* work.

1.5 Defining and Using a Personal Process

This book shows you how to define and use an operational personal process. However, the examples use my processes and not yours. When first experimenting with the PSP, I did not know what such a process should look like or how it would feel to actually use one. Over several years, I developed and scrapped many processes because they didn't work, were too complex, or they just didn't feel comfortable. After doing this for nearly three years, and after writing 72 programs in Pascal and C++, I finally got the hang of it. Process design and development is not all that hard if you approach it properly and if you have actually used such a process yourself.

This book shortcuts this entire process learning cycle by having you use my processes so you can see what it feels like to follow a defined, measured, planned, and quality-controlled personal process. It is surprisingly easy and the results are truly amazing.

Take the example of Maurice Greene, who broke the world record for the 100-meter race in Athens on June 16, 1999. Although he had always been a fast runner, he had not been winning races and was becoming discouraged. Then he went to see coach John Smith in Los Angeles. Smith videotaped Greene doing a sprint, and then broke Greene's ten-second run into 11 phases. He analyzed each phase and showed Greene how to maximize his performance in every one. This became Greene's defined and measured personal running process. After several months of coaching, Greene started winning world records, and for several years he was known as the fastest man alive.

How would you like to be the fastest software developer alive, or the best by any other measure you choose? Today, we have no way to even talk about this subject. Our work is essentially unmeasured. Like Greene, we are all trying harder but we don't know whether we are improving, or even what improvement would look like. That is what a defined, measured, planned, and quality-controlled personal process can do for you. Although you may not want to put in the months of

31422 01 pp. 001-010 r2ah.ps 2/9/05 5:21 PM Page 8

rigorous training needed to become a world champion, we all want to do our jobs in the best way we can. With a defined and measured process, not only will you know how well you are doing your job today, you will see how to improve, and to keep improving as long as you keep developing software.

So there it is. Like coach John Smith with Maurice Greene, I am asking you to use my process, at least until you see how such a process works. Although this process may not be the one you would design for yourself, once you have finished the PSP course, you will understand how these processes work and can design some of your own. For now, try to live with my processes and use them as well as you can. So far, thousands of developers have used them and many have continued to use them after finishing the PSP course.

1.6 Learning to Use a Personal Process

When you use this book in a PSP course, you will write several programs using the evolving process shown in Figure 1.1. The first process, PSP0, is a simple generic process that starts you writing programs much as you do today. The only addition

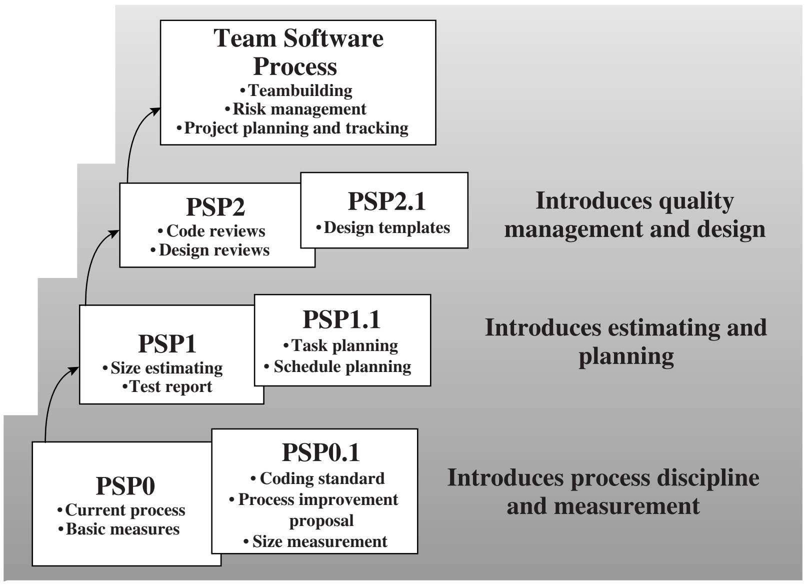

FIGURE 1.1 PSP PROCESS EVOLUTION

is time and defect measures. The PSP0 process is described in Chapter 2 and is used to write at least one program. Then, in Chapter 3, size measures are added to produce the PSP0.1 process that you use to write one or more additional programs. As you will see, size measures are essential for understanding how well you work. For Maurice Greene, for example, the size measure was the length of the track. Without that measure, he would not have known how fast he ran or if his times were improving.

Next, Chapters 4, 5, 6, and 7 show you how to use historical size and time data to estimate program size and development time. These chapters introduce the PSP1 and PSP1.1 processes that you use to write more programs. Chapter 8 covers quality, and Chapter 9 describes the PSP design and code reviews. With PSP2, you then write one or two more programs. Finally, in Chapters 10, 11, and 12, you write several more programs using the PSP2.1 process and the PSP design and quality methods. These methods show you how to develop essentially defect-free programs that you can test in a tenth of the time required for programs developed with prior methods.

1.7 Preparing for the Team Software Process

Once you finish this book and work through the program and report exercises, you will be prepared to work with other PSP-qualified professionals on a TSP team. The PSP shows you how to do quality work for predictable costs on committed schedules. This is software engineering. Once software developers have taken the PSP course and learned how to be software engineers, most of them end up working on teams that use the TSP. The rest of this book describes the PSP. Finally, Chapter 14 describes the TSP and shows how the PSP prepares you to be a TSP team member.

1.8 Summary

A personal process is something that you use to guide your work. With the PSP concepts and methods in this book, you can determine how well your current process works for you and learn where and how to improve. With the PSP, you will make accurate plans, consistently meet commitments, and deliver high-quality products. Although the exercises in this book concentrate on writing module-sized programs, that is only to help you learn PSP principles and methods. After completing this book, you should define and use your own personal processes for every aspect of your work, from requirements through test and product maintenance. Thousands of engineers have now been trained in the PSP and they have almost universally found that these methods help them to do much better work. They have also found that using these methods on a TSP team is a truly enjoyable and rewarding way to work. It will, in fact, change your life.

Reference

Deming, W. E. *Out of the Crisis.* Cambridge, MA: MIT Press, 1982.