**Chapter 12. Process Improvement**

**If you ask a dozen people on a software team** to describe how the team should build software, you'll get at least a dozen different answers—and most of those descriptions, if followed, will produce *something*. But what's produced may not be very good: it may not be useful to the users, it may not be of very high quality, and the team may not be comfortable building, delivering, or maintaining it. A software process makes sure that everyone on the team agrees up front on how they will build software, while simultaneously ensuring that the software will be built in a way that works for the team and the organization.

**Life Without a Software Process**

Many process improvement experts see the world as black and white: there are bad software teams without a formal process , and good teams that have one in place. But the world is not that simple. You should understand exactly what a software process brings to the table, and why having a process is important.

**Teams Can Be Effective Without a Formal Process**

There are many successful teams that do not spend time looking at the big picture and writing down a software process. In most organizations like this, individual programmers or small teams are responsible for entire projects: they talk with the users and stakeholders to understand what they need from the software, they build the software, and they deliver it to the users. People in this situation get used to the "jack-of-all-trades" role—they do everything themselves, from gathering requirements to designing and building the software to testing and deploying it. This is an effective way to build software: by limiting the number of people involved and making each individual responsible for the entire project, less communication and documentation are required, and there is less overhead and complication for the project.

In addition to the "jack-of-all-trades" situation, there are also "skunk works" programmers who will generally take the initiative and build software that was not necessarily asked for, but that addresses certain users' or clients' needs. This tends to happen when a programmer has a lot of knowledge about the needs of the users, and she can independently identify tasks that are being performed poorly that could be automated with software. When this goes well, the programmer is revered as a genius in the organization: she produced software "under the radar" that was useful to the people in the organization, yet required no work on their part: it was simply delivered, finished, as a wonderful surprise. This is a highly satisfying way for a programmer to work—no pressure, no expectations, and high rewards.

Another way that many teams work effectively is by relying on a development manager who has a very good understanding of the needs of the users and stakeholders. A highly capable manager who is willing to put an enormous amount of effort into design, architecture, and programming can serve as the hub of all software projects, parceling out work to individual programmers and integrating it all back together. As long as this person is capable of keeping everything in his head, the team can keep delivering software. This requires an enormous commitment on the part of the manager, but it is also very satisfying and leads to a lot of respect from others in the organization (not to mention job security).

What all these organizations have in common is that they do not have a formal software process. A *software process* is a set of activities that, if done, will result in software. It's important to recognize that all of the organizations described in the previous few paragraphs do have a software process—it's just not a *formal*, or documented and repeatable, process.

You are already familiar with many of the activities in typical software processes, because they include many of the tools and techniques in the first part of this book! In an organization that has put a lot of effort into a good process improvement program, the process activities will include a wide variety of tasks. These tasks will cover not just programming but also requirements management, project planning, configuration management, quality assurance, and other tasks meant to ensure that the software is of sufficient quality. Making improvements to the process will positively affect the way the team builds software in the future.

It's very important to recognize that teams that do not have a formal process are generally happy! A team that produces successful software projects can point to their successes with pride. The varied work, high visibility, and wide responsibility mean that the programmers are highly respected in the organization. Respected, that is, except when projects fail.

The most common source of failure is that this sort of team does not scale up very easily. While it's true that small projects and small teams can work well without a formal software process, it gets harder and harder to do the work as the scope of the software project gets wider, the team gets bigger, or the users and stakeholders become less available. When a team outgrows an informal process, projects begin to have problems. Programmers who used to produce lots of software find that their projects have started to feel "bogged down." The users are increasingly unhappy and the software is increasingly late. The work just doesn't seem fun anymore. When this happens, many of the "Diagnosing Problems" scenarios in the first part of this book start to look very familiar.

The shift from a happy and productive team to a bogged-down team usually happens because the team has taken on more people, larger projects, or new kinds of work. Suddenly, they have to retain more knowledge about the users' and stakeholders' needs than they can keep in their heads. Some teams find that this happens when they add the fourth (or sixth, or ninth) person to the team; others find that it happens when they try to take on a project that does not resemble any of their previous ones, or when they team people up to work on larger or more complex projects.

One of the most common situations in which a team gets "maxed out" arises when a small programming group with a good track record of building projects for individuals is faced with having to build a project on a larger scale—especially if it is intended to be used by people outside the organization. Previously, the programmers were able to sit down with the people who would be using the software on a day-to-day basis and talk with them about what they needed. Those users would stay involved with the project throughout the entire development phase, giving feedback on prototypes and intermediate builds and helping the programmers learn more about their needs. A programming team used to working in this environment can have trouble building software meant for users outside of the organization. The problem is that the users are no longer available to the programmer whenever input is needed. A programmer used to walking across the office and asking for clarification about a confusing or unclear concept will often have trouble adjusting to a situation in which that input is not readily available.

This problem is compounded when the knowledge needed to define the behavior of the software does not exist anywhere in the organization. Programmers know how to build software; they are not necessarily familiar with the day-to-day business of the people who will use it. This is where the "jack-of-all-trades" can suddenly find himself on shaky ground. He's spent a great deal of time learning about the particular needs of a small number of users; if a new project requires that he understand a completely different set of users, he may have to spend a very long time building up sufficient expertise in this new area.

**Lack of Process Maturity Is Not "Immature"**

There are many excellent textbooks on the nuts and bolts of software process improvement, but few of them acknowledge the fact that many successful organizations do not have a formal software process. In fact, process improvement experts often talk about process improvement in ways that are actually insulting to software professionals who feel they have done a good job and gone a long way in their careers. They use words such as "immature," "haphazard," "chaotic," and even "childish" to describe the software process of an organization that does not have a formal software process. This is not a useful way to convince people to change the way they do their jobs. (This was a major consideration in choosing "initial" as the label for the first maturity level in the CMM—see the section "[The Capability Maturity Model](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch12s02.html#appliedprojectmgmt-CHP-12-SECT-2.1.1).")

For example, one otherwise excellent book on practical process improvement contains a chapter demonstrating this problem. It compares the key behaviors of a child (moody, unpredictable, inconsistent, living for the moment, thrown into a panic by unexpected setbacks) with the behaviors of a "mature adult" (stable, reliable, consistently capable, plans ahead, copes well with the unexpected). The author uses this comparison to illustrate the difference between an organization with a formal process and one without. This is quite insulting to anyone who works in an organization without a formal process; it's also untrue, since there are plenty of organizations that in no way display those characteristics. Yet some process improvement professionals attempt to change organizations by showing them exactly this sort of "evidence," presumably in hope of shaming the team into adopting the changes they recommend.

Even worse than insulting people into changing is evangelizing to them, but this is exactly what many process improvement experts do. As unlikely as it sounds, process improvement textbooks and articles literally talk in terms of religion, believers, disbelievers, and evangelism. To someone who thinks this way, process improvement is a way of thinking that requires faith; unbelievers cannot be "converted," no matter how much they are cajoled or harangued. It's not surprising that many process improvement efforts fail. People distrust evangelists and suspect they have more of an allegiance to the process than to the organization and the team. Software engineers who are pestered or pressured into "accepting" a new process will not hesitate to abandon it if they are put under any external pressure to do so—especially if they had no input or feedback when it was developed.

This is no way to convince someone to change the way he does his job. But it is, unfortunately, a representative example of the way some people approach process improvement. They treat the programming team as a bunch of immature children who can barely do the work assigned to them. Or they treat the team members as infidels and unbelievers, leading the team to wonder how these religious lunatics got into the office. Then the experts are surprised that the team members—who consider themselves and their past projects successful, even if they weren't perfect—reject this new process improvement effort.

The truth is that programmers love being the "go-to" person. They love having lots of successes, and they love when users can go directly to them without having to bother with documentation and testing in order to get programs written. There is a lot of job satisfaction in doing things this way. And in cases in which the software is all in the same area of the organization's business and repeatedly draws on the same knowledge of the users' needs, a team can be highly successful working like this. If you put yourself in the programmers' shoes, you'll realize that you would need a really good reason before you would be willing to change. (See [Chapter 9](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch09.html) for more information on why changes like this often fail, and how to successfully make changes in your organization.)

**If Things Are So Great, Why Change?**

If there are no complaints about the way the team is building software, then there's no reason to change! If all your organization needs from you is that you assign one or a few programmers per project, and if you can always guarantee that experts, users, and stakeholders will be available to the programmer whenever she needs them, then it might make sense to keep things flexible and variable and change them only as needed.

However, few teams are really in this situation. At some point, most teams will need to grow in size or capability. This can mean taking on more programmers and solving more complex problems. Without a formal process, organizations often run into problems when they try to expand the team or get people to work together on larger projects. For example, the programmer who was working on one project could be needed on another team, or could leave the organization altogether; this makes it difficult to expand the team because they have lost critical knowledge required to build and maintain the software. If there is no formal process in place, there is no guarantee that knowledge was ever written down.

The most serious problems in expanding a team involve communicating the needs of the organization, users, and stakeholders, and setting the expectations of the organization about what will be delivered, when it will be delivered, and how much it will cost. Having a formal process in place ensures that there is consistency between projects, so an expanding team can rely on a library of past estimates, guidelines for working together, and other important tools to help them work. They do not have to keep reinventing the wheel; for every problem that has been solved before in the organization, the solution is available to them.

Expanding the team is not the only place a formal process is useful. It can also help in an organization where experts, users, or stakeholders are no longer readily available to the programmers. This is another area where the formal process can help. It means that requirements gathering activities are planned for from the beginning, and that there is a training program in place to help the users and stakeholders learn to work with the requirements analysts, ensuring that requirements are gathered at the beginning of the project.

# Software Process Improvement

Software process improvement is the art and science of changing an organization's software process in order to build better software. In the first part of this book, you learned to diagnose and fix problems for individual projects. When you adopt a specific tool for your project—for example, if you implement inspections on your project team and have them all follow the script in [Chapter 5](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch05.html)—you are formalizing part of the software process for that project by having the team follow a written description of that activity. Software process improvement is very similar, except that instead of improving one project at a time, you work on improving the entire organization.

The tools and techniques in the first part of this book make up many of the nuts and bolts of software process improvement . But while this book so far has been about making specific improvements to the way software is built on the scale of an individual project, there is another perspective: the high-level organizational perspective. It's very important to diagnose chronic problems your organization is having and address them with specific practices to be adopted for all projects. By stepping back and looking at the software process as a whole, you may be able to plan ahead and anticipate the inefficiencies in the way your team develops software. In doing so, you can avoid problems before they have serious impact on your organization. That is where software process improvement can really help your team excel.

Software process improvement always involves looking at the big picture. This generally means writing down the entire software process as a whole and making sure that it is followed on each project. It involves not just diagnosing specific problems and attacking them individually, but also looking at entire projects and identifying areas that can be improved.

Software process improvement differs from the approach described in the first part of this book. In [Part I](https://learning.oreilly.com/library/view/applied-software-project/0596009488/pt01.html), you learned how to diagnose individual problems on your projects and use specific tools, techniques, and practices to fix those problems. In much the same way, software process improvement can be evolutionary, allowing you to use iterative cycles to fix individual parts of your organization's lifecycle. But software process improvement differs from the diagnose-and-fix approach, because it requires that you change the culture of your organization to one of continuous process improvement. Software process improvement requires patient and consistent support (in both actions and resources) from your senior managers, in order to be effective. It also requires a broad consensus among the software engineers. If you can gather this kind of support behind a process improvement effort, you can address issues that are beyond the scope of individual projects and dramatically increase the capability of your software organization.

There are specific process improvement tools to help you do this. There are models and certifications that help you assess the state of your organization's process, and that serve as a framework for improving that process. There are also processes and methodologies you can adopt that describe the complete set of activities, roles, and work products needed to build software. By applying these tools, you can give your organization the ability to fix problems before they get serious enough to cause your projects to slow down.

## Models and Certifications

To many experts, software process improvement in practice means using a model or certification standard as a guideline for assessing and improving a software organization. Some of the most common models are the Capability Maturity Model, ISO 9000, and Six Sigma. These are not processes in and of themselves. Rather, they are systematic frameworks that were developed for evaluating any software organization (no matter what process is in use), identifying improvements that will increase the ability of the organization to build better software, and certifying that the organization has met an objective standard for capability.

### THE CAPABILITY MATURITY MODEL

The Capability Maturity Model (CMM), is a process improvement method developed by the Software Engineering Institute at Carnegie Mellon University. It provides a set of best practices meant to address important aspects of software development: productivity, performance, costs, predictability, and stakeholder satisfaction. The purpose of the CMM is to define the characteristics of a mature, capable process in a way that can be measured and compared to processes at other organizations.

The CMM was developed in coordination with the U.S. Department of Defense (DoD) to help organizations privately improve their own processes. (It was also adopted by the DoD as a way to provide the U.S. military with a consistent method to identify the most capable software contractors. While this has been successful, it has also led to some abuses of the assessment system—see below.) The result was a model (first published in August 1991) that contained five maturity levels, ranging from "Initial" (in which an organization does not apply tools, techniques, and practices consistently across all of its projects) to "Optimizing" (in which a software process was well defined, brought under statistical control using various metrics and measurements, and continually improved based on those metrics).

By the late 1990s, the government and the CMM user community had a great deal of feedback for the SEI team responsible for maintaining the CMM. In response, a project to create the Capability Maturity Model Integration (CMMI) was initiated in order to address these concerns. This new version was released in 2002.

Both the CMM and CMMI are divided into key process areas (KPAs) that define specific goals and practices. Each KPA addresses a specific area of software engineering. There are several dozen KPAs, including requirements management, project planning, project monitoring, configuration management, and training. (These areas of improvement should seem familiar by now!)

Each of the five maturity levels of the CMM is represented by its own set of KPAs; for an organization to achieve a specific maturity level, it must implement all of the practices in each of the KPAs for that level. The CMMI also contains five maturity levels; however, some of the focus of the CMMI has been drawn away from maturity levels in favor of implementing specific practices and goals. It does, however, still support a level-based assessment.

The most important part of each KPA is a set of one or more goals that define the "heart and soul" of each KPA. For example, the Project Planning KPA of CMMI v1.1 contains three goals:

Establish Estimates

Estimates of project planning parameters are established and maintained.

Develop a Project Plan

A project plan is established and maintained as the basis for managing the project.

Obtain Commitment to the Plan

Commitments to the project plan are established and maintained.

These are sensible goals that, if met, will lead to a well-planned project; if any of these goals is not met, then the project will have a clear planning problem. The remainder of the KPA lists specific practices that should be performed, in order to meet those goals. For example, the "Establish Estimates" goal is met with specific practices:

Estimate the Scope of the Project

Establish a top-level work breakdown structure (WBS) to estimate the scope of the project.

Establish Estimates of Work Product and Task Attributes

Establish and maintain estimates of the attributes of the work products and tasks.

Define Project Life Cycle

Define the project life cycle phases upon which to scope the planning effort.

Each practice is further defined with subpractices. For example, the "Estimate the Scope of the Project" practice requires that a WBS is developed, work packages are defined in sufficient detail to provide estimates, and work products are identified that will be acquired externally or reused. Again, these are all sensible activities that need to be performed in order to estimate the scope of a project.

Within each practice and subpractice is additional information providing advice and tips for implementation. There are usually lists of typical work products—for example, the subpractice for estimating the scope of a project lists the task descriptions and WBS as work products. However, these are just suggestions. The most important part of the CMM and CMMI is meeting the goals, and the practices are simply an efficient way of meeting those goals.

When an organization begins a process improvement effort based on the CMM or CMMI, a good first step is to set up a software engineering process group (SEPG). This is a team of software engineering professionals and/or managers within the organization who are dedicated either part- or full-time to improving the software process. They identify problems and inefficiencies, define the practices needed to address those problems, and implement them in the project teams.

Another important part of the CMM and CMMI is the assessment process. In an assessment, an assessor is brought into the organization. He spends time (typically one week) reviewing the process documentation, interviewing the people on the team, and verifying whether the process is actually carried out on real projects. The result of the assessment is a report that identifies which practices are in place, which practices are missing from key process areas, and the maturity level of the organization.

Assessments are a useful tool that SEPGs can use to identify areas for improvement. However, there is a downside to assessments. There are many organizations that must comply with the CMM or CMMI in order to qualify for certain government programs or to bid on contracts that require a minimum maturity level. Other organizations pursue a maturity level for publicity or public relations purposes. The assessment process assumes that the organization being assessed is honestly representing its practices. A software organization that is intent on getting assessed at a certain level can create reports and work products that will make it appear as if all of the practices have been implemented. This is just like a company that fools an accounting auditor by keeping two sets of books—it may pass the audit through no fault of the auditor.

When the CMM or CMMI is abused, it is no longer an effective way to improve an organization. The practices that are part of the KPAs then simply amount to pushing paper around. Documents that could be useful have simply become bureaucratic forms to be filled out. A project plan is created not to help the team understand and plan the work, but simply to satisfy an organizational policy. When this happens, the team members still face the same problems they have always faced: increasing delays, decreasing quality, overtime, dissatisfaction, and poor software development practices. But now they also get to fill out a bunch of paperwork.

Despite this potential abuse, when the CMM or CMMI is used well, it can be a powerful tool for actually improving the software process and capability of an organization. It contains a wealth of good practices, as well as advice for implementing those practices. When an organization really works toward understanding and fulfilling the goals and implementing the practices, they stand a good chance at truly building better software.

### NOTE

The Capability Maturity Model and a great deal of supporting information can be obtained for free from the Software Engineering Institute web site at <http://www.sei.cmu.edu/cmmi/cmmi.html>.

### ISO 9000

ISO 9000 is a family of quality management standards defined by the International Standards Organization and implemented by over half a million organizations around the world. Quality management refers to the practices performed by an organization in order to fulfill the customer's requirements (and any legal or regulatory requirements). The goal of quality management is to improve customer satisfaction, while at the same time continually improving the performance of the organization.

Every ISO 9000 standard defines a set of minimum "pass or fail" standards that are used to judge whether an organization is in compliance. ISO standards, like the CMM, have a certification process in which an organization's practices are assessed by a third-party assessor who audits the organization's compliance with the quality system, and whether that system is effective. The result of the audit is a set of recommendations for changes to be made, in order to bring the organization into compliance.

ISO 9000 is based on eight core principles:

* Organizations must focus on their customers by understanding current and future customer needs.
* Leaders within the organization must create and maintain an environment in which people can become involved and fulfill the organization's objectives.
* People at all levels are important to the organization.
* Activities and resources are best managed as a process.
* Organizations have many interrelated processes, which must be understood and managed as a system.
* The organization should continually improve its performance.
* Decisions should be well informed and based on real data and information.
* An organization and its suppliers are in a mutually beneficial relationship.

The ISO 9000-3 standard contains a set of guidelines that interprets ISO 9000 so that it can be applied to the development, supply, and maintenance of software. It is divided into sections that define standards for many areas of a software organization, including management practices, the quality system, contracts, document and data control, inspection, training, deployment, process control, and the design and development of the software.

Each of the sections contains standards for the day-to-day work that goes on in the organization. For example, within the software development and design section are standards for software development, software design, design and development planning, organizational and technical interfaces and design review, verification, validation, and change control.

Each of these standards defines specific practices that must be implemented in the organization. For example, the software development requirements require that a project plan be developed. This plan must define the project, list its objectives, contain a project schedule, define the inputs and outputs, identify related plans and projects, identify project risks, and identify assumptions. These requirements should seem familiar: they are all part of the project plan in [Chapter 2](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch02.html).

Like the CMM, the ISO standards can be abused by an organization and reduced to bureaucracy and paper pushing. There is a great deal of pressure on many organizations to achieve an ISO 9000 certification at any cost, since they may be ineligible for certain contracts without it. However, when the standards are used properly, they can provide good guidelines to a project manager looking to improve how her organization builds software.

### NOTE

Information on ISO 9000 can be found at the International Standards Organization web site. The URL at the time of this writing is <http://www.iso.ch/iso/en/iso9000-14000/iso9000/iso9000index.html>

### SIX SIGMA

Six Sigma is an approach to improving quality in manufacturing and business processes. It was developed at Motorola in the early 1980s, and it has been used successfully at many software organizations. The goal of Six Sigma is to produce a product of consistent quality by statistically measuring the defect rate, improving the processes to eliminate those defects, and then monitoring the improvements. Six Sigma has been used to successfully improve organizations in many industries. While it has a reputation for success in large companies with thousands of employees, Six Sigma can be applied to small project teams as well.

The Greek letter sigma refers to standard deviation—Six Sigma means "six standard deviations from the mean." To achieve this level of quality in a manufacturing process, 99.9997% of all products must be of acceptable quality (or 3.4 defects per million opportunities). It's not hard to get an intuitive grasp on what this means in the real world. According to Jeannine Siviy at the Software Engineering Institute, a Four Sigma level of quality—meaning that we're 99.9% sure—would yield:

* 9 hours of unsafe drinking water a year
* 107 incorrect medical procedures a day
* 200,000 incorrect drug prescriptions a year
* 18,322 pieces of mishandled mail an hour
* 2,000,000 documents lost by the IRS a year
* Two short or long landings at any major airport a day

| **Phase** | **Description** |
| --- | --- |
| **D** efine opportunities | Determine customer and core processes . Determine the customer's requirements for the  products and services being produced. Map the processes that are being improved.  Gain the customer's commitment. |
| **M** easure performance | Develop a plan to collect and measure the defect data.  Collect data from many sources in the organization and determine the  defect rates and other metrics. Compile and display the data. |
| **A** nalyze opportunity | Analyze and verify the data collected. Determine the root causes for the defects  and identify opportunities for improvement. Prioritize the improvement opportunities. |
| **I** mprove performance | Design creative solutions to improve the processes.  Create a problem statement and a solution statement for each problem.  Test specific improvements with an experimental approach. Deploy the improvements. |
| **C** ontrol performance | Monitor the improvement programs to control them.  Develop an ongoing monitoring plan to keep the process on the new course  and prevent it from reverting to its previous state.  Assess the effectiveness of the improvement.  Develop staffing and training incentives to make the improvements permanent. |

Variance is an important part of Six Sigma. There is a saying at General Electric, an early adopter and innovator in the Six Sigma world: "Our Customers Feel the Variance, Not the Mean." All processes have some inherent variability. No process, especially not a software process, produces defects at an entirely regular rate. What GE found is that customers can get used to a "noise level" of defects; it's the large changes in quality that will really get their attention. This is where Six Sigma is especially useful. It is the statistical variances that will take a product from one standard deviation into another; the process problems that cause these variances are the ones most likely to be identified and fixed.

The goal of Six Sigma is to think of every aspect of the business as a process that can be improved in a way that can be measured statistically. The main tool for doing this is a five-phase approach called DMAIC (see [Table 12-1](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch12s02.html#appliedprojectmgmt-CHP-12-TABLE-1)).

*Table 12-1. DMAIC: A five-phase approach to Six Sigma*

Six Sigma professionals have a training and certification program called the Black Belt program. The Certified Six Sigma Black Belt is a professional trained to implement Six Sigma in an organization and train others in the Six Sigma principles, systems, and tools. They have a thorough understanding of the DMAIC model and fundamental knowledge of project management.

### NOTE

More information on Six Sigma can be found at the iSixSigma web site ([http://www.isixsigma.com](http://www.isixsigma.com/)) and the Motorola University web site (<http://www.motorola.com/motorolauniversity>).

## Processes and Methodologies

There are also complete processes that can be adopted by an organization. Unlike models and certifications, these methodologies define the activities to be performed and the roles that people in the organization must fill in their daily work. In many cases, they incorporate many project management practices similar to the ones in this book. But unlike the diagnose-and-fix approach, a methodology provides a complete process that can be adopted all at once by an organization.

Adopting an "off-the-shelf" process or methodology is fundamentally different from using a model or framework for process improvement. Instead of improving the existing software lifecycle, you adopt an entire process that's "tried and true." Unlike the diagnose-and-fix approach, adopting a specific process requires revolutionary, rather than evolutionary or incremental, changes. It is usually much harder to convince people to adopt an entire new process rather than one specific tool or technique. It also requires determined management support to gain acceptance, because it requires a complete change to the entire lifecycle.

### EXTREME PROGRAMMING

Extreme Programming (or XP) was developed in the 1990s by Kent Beck when he was working on a project at Chrysler (although its roots stretch back to SmallTalk projects that he and Ward Cunningham worked on at Tektronix in the mid-1980s). It represents one of the fastest growing movements in the software world today, and many of the techniques in this book will seem familiar to people who have worked in an XP environment.

XP consists of a set of rules and practices that govern all areas of software development: planning, designing, coding, and testing. These practices emphasize interaction and collaboration between the engineering team and the stakeholders and users, as well as the ability to respond quickly to changes in order to produce working software. The goal of XP is to lower the cost of change. Uncontrolled changes are the most common cause of software project failure (see [Chapter 6](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch06.html)); by putting basic XP principles and practices in place, a project team can control the changes.

The specific planning practices employed by XP are intended to be as lightweight and agile as possible, and have a high level of user involvement. XP projects are planned in short iterations using the Planning Game (see [Chapter 3](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch03.html)) in order to make frequent small releases. The requirements are documented as user stories, which are written by the users. Each user story is a brief, three-sentence description of a specific behavior that must be performed by the software. It is much less detailed than a use case or a requirement, and should only contain enough information to allow a developer to create a basic 1, 2, or 3-week estimate. Once 80 or so user stories are created, a release plan is built by the developers; they divide the project into 1- to 3-week iterations. An iteration plan is developed when each iteration begins. Project velocity is a metric that gauges how much work is getting done on the project by comparing the sum of the estimates for the user stories planned in the current iteration with the estimates for the tasks actually performed (similar to variance, in [Chapter 4](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch04.html)).

The main principle behind XP design is simplicity: a simple design is always more efficient to build than a complex one. Classes and methods should be named consistently; a metaphor should be used to help guide developers in creating consistent names. Functionality should never be added before it is scheduled. The developers should "refactor mercilessly"—that is, they should refactor the design and code whenever possible (see [Chapter 7](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch07.html)).

XP coding employs some familiar practices: test-driven development (see [Chapter 7](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch07.html)) and pair programming (see [Chapter 5](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch05.html)). In addition, one important principle in XP coding is that the customer must always be available to the developers. This allows them to get constant feedback and clarifications. Users and stakeholders are an active part of the development team, and the developers have face-to-face communication with them throughout the course of the project. In addition to the user stories, they are responsible for negotiating which user stories are included in each release, giving feedback on the results of each iteration and, most importantly, filling in the details that are missing from the user stories.

Testing in XP consists of unit tests and acceptance tests. All code must have unit tests, and all unit tests must pass before the code can be released. When a bug is found in production, an additional unit test must be written to guard against it in the future. When software is delivered, the customer is responsible for running acceptance tests to verify that the user stories are all implemented properly. The customer is responsible for verifying that the acceptance tests are correct—meaning that the unit tests truly verify that the work has been done—and for reviewing the results of the acceptance tests to determine whether the software is ready for release.

Many programmers have turned their struggling development teams around using XP. One of its biggest advantages is that it does not require widespread organizational change, which means that it can be implemented by a small group of programmers without getting managers involved. For a team that does not have any project planning or requirements gathering, working in an XP environment will be an enormous relief. XP is an effective way to promote good social and organizational change by putting good development and project management practices in place. It's no wonder that there are a growing number of professionals who have come to depend on XP for all of their work.

One common misunderstanding is that XP is not a disciplined or well-defined process, and that XP and the CMM and ISO 9000 are all somehow mutually exclusive. In fact, XP projects follow a very specific set of rules and practices. It is possible for a good team that has fully implemented XP to pass a CMM or ISO 9000 assessment.

On reason XP is effective is that it addresses one of the most common problems in software development: the hands-off customer. Just as a hands-off client can cause serious problems in an outsourced project (see [Chapter 11](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch11.html)), serious project problems can happen when a stakeholder does not act as if he has a real stake in the software. It is very easy for a software project to go off track, and it needs constant direction to keep the scope of the project current with the organization's needs. XP provides an effective solution to this problem by making the stakeholders a part of the development team and requiring them to be involved in the day-to-day work of the project.

There are, however, some important drawbacks to XP. It is not clear that XP can be extended to large teams. Some experts feel that XP is difficult to implement with teams that are larger than about 12 people. (There is some research in this area, and a few researchers have reported success on large teams using a modified version of XP.) Another drawback is that it only works on software projects that can be delivered incrementally, or do not have human users. The original C3 project at Chrysler was a payroll system; it's not clear that XP would work for a different team at Chrysler developing a fuel injection system or anti-lock brake system.

But the most important drawback to XP is that the customer must always be available. There are many environments in which this is a very difficult requirement to meet. The users and stakeholders in most organizations have their own jobs to do, and it would be unreasonable to require that they devote all of their working time to a software project. Requirements elicitation (see [Chapter 6](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch06.html)) requires specialized skills and people, which often makes it difficult to implement in small organizations, but it is much less demanding of the time of people outside of the engineering team. And once the requirements are written, they can be reviewed for defects. While it's generally not possible to catch every defect before the programming begins, it is possible to find many of them—and it is much faster to fix defects on paper than it is to fix them in software, especially when they are not caught until late in the project.

XP proponents often say that face-to-face communication is superior to written communication, and that XP's reliance on face-to-face meetings and very little documentation frees the process from the "bureaucracy" of requirements documentation. However, reliance on face-to-face communication for detailed requirements introduces its own problems. Defining the behavior of software is a particularly complex and error-prone process. It is very easy for people to reach misunderstandings when talking about software behavior. This is especially risky when a customer has been involved as a project team member since the beginning of the project. The customer's expectations will often evolve along with the project, as a side effect of the influence of the rest of the programming team. The result is that the customer is more likely to accept the software, even if it does not fully meet the needs of the people in the customer's organization who were not on the project team. This sort of tunnel vision is much easier to avoid when the customer keeps his eye on his task and his organization's needs rather than on the details of the project and the developers.

Drawbacks aside, for projects in which the circumstances and team are compatible with XP, it is a very powerful tool. Many project teams across the world have made a huge difference in their projects by adopting some or all of the XP practices.

### NOTE

Additional information about Extreme Programming can be found in Extreme Programming Explained: Embrace Change(2nd Edition) by Kent Beck (Addison Wesley, 2004).

### RATIONAL UNIFIED PROCESS

There are many complete software processes that are available and ready to be adopted out of the box. Typically, such a process will include a complete set of activities to be performed over the course of the entire software project. These activities usually include scope definition, requirements engineering, architecture, design, programming, and testing activities. One of the most popular off-the-shelf processes is the Rational Unified Process (RUP).

The activities of RUP are based around the idea of highly iterative development. After an initial planning phase, the software project enters a cycle of iterations, each of which results in an executable release. Each of the iterations contains distinct activities for planning, requirements specification, analysis and design, implementation, testing, and stakeholder evaluation. The advantage of iteration is that it allows the project team to identify and correct misunderstandings early on in the project, keep the stakeholders up-to-date with deliverables to help them gauge the progress of the project, and distribute the project's workload more evenly over the course of the project.

RUP includes a disciplined approach to requirements management that is based on the idea of managing changes. The requirements in RUP are similar to those in [Chapter 6](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch06.html). They include use cases as well as functional and nonfunctional requirements. RUP incorporates software design using the Unified Modeling Language (UML), a visual modeling system for graphically representing the use cases, class model, object interactions, and components of the software.

One core element of RUP is the continuous assessment of the quality of the system. The testing activities involve creating test cases based on the functionality introduced in each iteration. By incorporating testing activities in each iteration, RUP allows the team to identify defects early, and to continuously assess the health of the product. RUP also has a clear change control system to ensure that every change is evaluated; measurements of the rate of software changes are used to assess the status of the project.

One thing that makes RUP unique is that it is a product in addition to a process. RUP is available from Rational Software, a subsidiary of IBM. An organization licenses RUP from Rational Software, which keeps the software up to date and provides regular updates to the process and its documentation.

There are many software packages that go along with RUP. Requirements and use cases are stored, managed, traced, and edited using the RequisitePro software. Rational Rose, a UML editing tool, is used to capture and visually model the software architecture. Other products are used for configuration management (ClearCase), change request management (ClearQuest), test case management (TestManager), performance testing (PerformanceStudio), test automation (Rational Robot), and tools to support other areas of the software process. Each of these products is also available separately, and RUP can be implemented without these tools. Many organizations license RUP as a complete process engineering suite. RUP was also built to be adaptable to the needs of individual organizations; Rational Process Workbench is a tool for customizing and extending RUP to meet your organization's needs.

The support tools in RUP are a strong feature. However, they also present a significant pitfall. Some project managers feel that they can simply buy their way out of a software process problem by buying RUP and distributing it to the team. But while some problems can be solved by adopting a process like RUP out of the box, many project management problems cannot be solved by writing a check. This is why it is often easier to adopt small, piecemeal changes like adopting individual practices, tools, and techniques than it is to force an entire organization to adopt a complete software process immediately. However, if that problem can be overcome, then an out-of-the-box process such as RUP can be a powerful tool for building better software.

# Moving Forward

There is no single process, framework, or methodology that works all the time. Every organization is different. What's more, organizations are made up of people who need to be convinced that the change really is necessary. While these practices can be very useful as guidelines, the most important part of improving the software process is helping people in your organization understand their role in the development of software.

One reason these process improvement tools have a good chance of successful implementation is that each of them takes into account the basic project management principles introduced in [Chapter 1](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch01.html). Whether the process is adopted all at once or good practices are adopted individually, if these principles are followed, then the team will build better software.

This book has been all about specific tools and techniques that, if applied individually, will help a project manager improve the way her organization builds software. Most software organizations will see substantial benefits from a piecemeal approach to building better software. Specific problems can be diagnosed, and individual tools and techniques can be applied to help alleviate those problems. In this way, a project manager can introduce changes directed specifically at the most troublesome problems. But whether you select a piecemeal, diagnose-and-fix approach or have the clout to initiate a full software process improvement effort, solving the problems that affect your projects is within your reach.