Examine the result. You can't be sure that your solution is perfect, but you can be sure that you've designed a sufficient number of tests to uncover as many errors as possible.

- Is it possible to test each component part of the solution? Has a reasonable testing strategy been implemented?
- Does the solution produce results that conform to the data, functions, and features that are required? Has the software been validated against all stakeholder requirements?

It shouldn't surprise you that much of this approach is common sense. In fact, it's reasonable to state that a commonsense approach to software engineering will never lead you astray.

1.5.2 General Principles

The dictionary defines the word *principle* as "an important underlying law or assumption required in a system of thought." Throughout this book I'll discuss principles at many different levels of abstraction. Some focus on software engineering as a whole, others consider a specific generic framework activity (e.g., **communication**), and still others focus on software engineering actions (e.g., architectural design) or technical tasks (e.g., write a usage scenario). Regardless of their level of focus, principles help you establish a mind-set for solid software engineering practice. They are important for that reason.

David Hooker [Hoo96] has proposed seven principles that focus on software engineering practice as a whole. They are reproduced in the following paragraphs:¹³

The First Principle: The Reason It All Exists

A software system exists for one reason: to provide value to its users. All decisions should be made with this in mind. Before specifying a system requirement, before noting a piece of system functionality, before determining the hardware platforms or development processes, ask yourself questions such as: "Does this add real value to the system?" If the answer is "no," don't do it. All other principles support this one.

The Second Principle: KISS (Keep It Simple, Stupid!)

Software design is not a haphazard process. There are many factors to consider in any design effort. *All design should be as simple as possible, but no simpler*. This facilitates having a more easily understood and easily maintained system. This is

Before beginning a software project, be sure the software has a business purpose and that users perceive value in it.

ADVICE 1

¹³ Reproduced with permission of the author [Hoo96]. Hooker defines patterns for these principles at http://c2.com/cgi/wiki?SevenPrinciplesOfSoftwareDevelopment.



"There is a certain majesty in simplicity which is far above all the quaintness of wit."

Alexander Pope (1688–1744)



If software has value, it will change over its useful life. For that reason, software must be built to be maintainable. not to say that features, even internal features, should be discarded in the name of simplicity. Indeed, the more elegant designs are usually the more simple ones. Simple also does not mean "quick and dirty." In fact, it often takes a lot of thought and work over multiple iterations to simplify. The payoff is software that is more maintainable and less error-prone.

The Third Principle: Maintain the Vision

A clear vision is essential to the success of a software project. Without one, a project almost unfailingly ends up being "of two [or more] minds" about itself. Without conceptual integrity, a system threatens to become a patchwork of incompatible designs, held together by the wrong kind of screws. . . . Compromising the architectural vision of a software system weakens and will eventually break even the well-designed systems. Having an empowered architect who can hold the vision and enforce compliance helps ensure a very successful software project.

The Fourth Principle: What You Produce, Others Will Consume

Seldom is an industrial-strength software system constructed and used in a vacuum. In some way or other, someone else will use, maintain, document, or otherwise depend on being able to understand your system. So, *always specify, design, and implement knowing someone else will have to understand what you are doing.* The audience for any product of software development is potentially large. Specify with an eye to the users. Design, keeping the implementers in mind. Code with concern for those that must maintain and extend the system. Someone may have to debug the code you write, and that makes them a user of your code. Making their job easier adds value to the system.

The Fifth Principle: Be Open to the Future

A system with a long lifetime has more value. In today's computing environments, where specifications change on a moment's notice and hardware platforms are obsolete just a few months old, software lifetimes are typically measured in months instead of years. However, true "industrial-strength" software systems must endure far longer. To do this successfully, these systems must be ready to adapt to these and other changes. Systems that do this successfully are those that have been designed this way from the start. *Never design yourself into a corner*. Always ask "what if," and prepare for all possible answers by creating systems that solve the general problem, not just the specific one. ¹⁴ This could very possibly lead to the reuse of an entire system.

¹⁴ This advice can be dangerous if it is taken to extremes. Designing for the "general problem" sometimes requires performance compromises and can make specific solutions inefficient.

The Sixth Principle: Plan Ahead for Reuse

Reuse saves time and effort. ¹⁵Achieving a high level of reuse is arguably the hardest goal to accomplish in developing a software system. The reuse of code and designs has been proclaimed as a major benefit of using object-oriented technologies. However, the return on this investment is not automatic. To leverage the reuse possibilities that object-oriented [or conventional] programming provides requires forethought and planning. There are many techniques to realize reuse at every level of the system development process. . . . *Planning ahead for reuse reduces the cost and increases the value of both the reusable components and the systems into which they are incorporated*.

The Seventh principle: Think!

This last principle is probably the most overlooked. *Placing clear, complete* thought before action almost always produces better results. When you think about something, you are more likely to do it right. You also gain knowledge about how to do it right again. If you do think about something and still do it wrong, it becomes a valuable experience. A side effect of thinking is learning to recognize when you don't know something, at which point you can research the answer. When clear thought has gone into a system, value comes out. Applying the first six principles requires intense thought, for which the potential rewards are enormous.

If every software engineer and every software team simply followed Hooker's seven principles, many of the difficulties we experience in building complex computer-based systems would be eliminated.

1.6 SOFTWARE MYTHS

vote:

"In the absence of meaningful standards, a new industry like software comes to depend instead on folklore."

Tom DeMarco

Software myths—erroneous beliefs about software and the process that is used to build it—can be traced to the earliest days of computing. Myths have a number of attributes that make them insidious. For instance, they appear to be reasonable statements of fact (sometimes containing elements of truth), they have an intuitive feel, and they are often promulgated by experienced practitioners who "know the score."

Today, most knowledgeable software engineering professionals recognize myths for what they are—misleading attitudes that have caused serious problems for managers and practitioners alike. However, old attitudes and habits are difficult to modify, and remnants of software myths remain.

¹⁵ Although this is true for those who reuse the software on future projects, reuse can be expensive for those who must design and build reusable components. Studies indicate that designing and building reusable components can cost between 25 to 200 percent more than targeted software. In some cases, the cost differential cannot be justified.

WebRef

The Software Project Managers Network at www.spmn.com can help you dispel these and other myths. **Management myths.** Managers with software responsibility, like managers in most disciplines, are often under pressure to maintain budgets, keep schedules from slipping, and improve quality. Like a drowning person who grasps at a straw, a software manager often grasps at belief in a software myth, if that belief will lessen the pressure (even temporarily).

Myth: We already have a book that's full of standards and procedures for

building software. Won't that provide my people with everything they

need to know?

Reality: The book of standards may very well exist, but is it used? Are soft-

ware practitioners aware of its existence? Does it reflect modern software engineering practice? Is it complete? Is it adaptable? Is it streamlined to improve time-to-delivery while still maintaining a focus on quality? In many cases, the answer to all of these questions

is "no."

Myth: If we get behind schedule, we can add more programmers and catch up

(sometimes called the "Mongolian horde" concept).

Reality: Software development is not a mechanistic process like manufactur-

ing. In the words of Brooks [Bro95]: "adding people to a late software project makes it later." At first, this statement may seem counterintuitive. However, as new people are added, people who were working must spend time educating the newcomers, thereby reducing the amount of time spent on productive development effort. People can be added but only in a planned and well-

coordinated manner.

Myth: If I decide to outsource the software project to a third party, I can just

relax and let that firm build it.

Reality: If an organization does not understand how to manage and control

software projects internally, it will invariably struggle when it out-

sources software projects.

Customer myths. A customer who requests computer software may be a person at the next desk, a technical group down the hall, the marketing/sales department, or an outside company that has requested software under contract. In many cases, the customer believes myths about software because software managers and practitioners do little to correct misinformation. Myths lead to false expectations (by the customer) and, ultimately, dissatisfaction with the developer.

Myth: A general statement of objectives is sufficient to begin writing

programs—we can fill in the details later.

Reality: Although a comprehensive and stable statement of requirements is

not always possible, an ambiguous "statement of objectives" is a recipe for disaster. Unambiguous requirements (usually derived

ADVICE 1

Work very hard to understand what you have to do before you start. You may not be able to develop every detail, but the more you know, the less risk you take. iteratively) are developed only through effective and continuous communication between customer and developer.

Myth: Software requirements continually change, but change can be easily

accommodated because software is flexible.

Reality: It is true that software requirements change, but the impact of

change varies with the time at which it is introduced. When requirements changes are requested early (before design or code has been started), the cost impact is relatively small. ¹⁶ However, as time passes, the cost impact grows rapidly—resources have been committed, a design framework has been established, and change can cause upheaval that requires additional resources and major design

modification.



Whenever you think, we don't have time for software engineering, ask yourself, "Will we have time to do it over again?" **Practitioner's myths.** Myths that are still believed by software practitioners have been fostered by over 50 years of programming culture. During the early days, programming was viewed as an art form. Old ways and attitudes die hard.

Myth: Once we write the program and get it to work, our job is done.

Reality: Someone once said that "the sooner you begin 'writing code,' the

longer it'll take you to get done." Industry data indicate that between 60 and 80 percent of all effort expended on software will be expended after it is delivered to the customer for the first time.

Myth: Until I get the program "running" I have no way of assessing its quality.

Reality: One of the most effective software quality assurance mechanisms

can be applied from the inception of a project—the technical review. Software reviews (described in Chapter 15) are a "quality filter" that have been found to be more effective than testing for finding certain

classes of software defects.

Myth: The only deliverable work product for a successful project is the working

program.

Reality: A working program is only one part of a software configuration that

includes many elements. A variety of work products (e.g., models, documents, plans) provide a foundation for successful engineering

and, more important, guidance for software support.

Myth: Software engineering will make us create voluminous and unnecessary

documentation and will invariably slow us down.

Reality: Software engineering is not about creating documents. It is about

creating a quality product. Better quality leads to reduced rework.

And reduced rework results in faster delivery times.

¹⁶ Many software engineers have adopted an "agile" approach that accommodates change incrementally, thereby controlling its impact and cost. Agile methods are discussed in Chapter 3.

Many software professionals recognize the fallacy of the myths just described. Regrettably, habitual attitudes and methods foster poor management and technical practices, even when reality dictates a better approach. Recognition of software realities is the first step toward formulation of practical solutions for software engineering.

1.7 How It All Starts

Every software project is precipitated by some business need—the need to correct a defect in an existing application; the need to adapt a "legacy system" to a changing business environment; the need to extend the functions and features of an existing application; or the need to create a new product, service, or system.

At the beginning of a software project, the business need is often expressed informally as part of a simple conversation. The conversation presented in the sidebar is typical.

SafeHome¹⁷



How a Project Starts

The scene: Meeting room at CPI
Corporation, a (fictional) company that makes consumer
products for home and commercial use.

The players: Mal Golden, senior manager, product development; Lisa Perez, marketing manager; Lee Warren, engineering manager; Joe Camalleri, executive VP, business development

The conversation:

Joe: Okay, Lee, what's this I hear about your folks developing a what? A generic universal wireless box?

Lee: It's pretty cool . . . about the size of a small matchbook . . . we can attach it to sensors of all kinds, a digital camera, just about anything. Using the 802.11g wireless protocol. It allows us to access the device's output without wires. We think it'll lead to a whole new generation of products.

Joe: You agree, Mal?

Mal: I do. In fact, with sales as flat as they've been this year, we need something new. Lisa and I have been doing a little market research, and we think we've got a line of products that could be big.

Joe: How big . . . bottom line big?

Mal (avoiding a direct commitment): Tell him about our idea, Lisa.

Lisa: It's a whole new generation of what we call "home management products." We call 'em *SafeHome*. They use the new wireless interface, provide homeowners or small-business people with a system that's controlled by their PC—home security, home surveillance, appliance and device control—you know, turn down the home air conditioner while you're driving home, that sort of thing.

Lee (jumping in): Engineering's done a technical feasibility study of this idea, Joe. It's doable at low manufacturing cost. Most hardware is off-the-shelf. Software is an issue, but it's nothing that we can't do.

Joe: Interesting. Now, I asked about the bottom line.

Mal: PCs have penetrated over 70 percent of all households in the USA. If we could price this thing right, it could be a killer-App. Nobody else has our wireless box . . . it's proprietary. We'll have a 2-year jump on the competition. Revenue? Maybe as much as 30 to 40 million dollars in the second year.

Joe (smiling): Let's take this to the next level. I'm interested.

¹⁷ The *SafeHome* project will be used throughout this book to illustrate the inner workings of a project team as it builds a software product. The company, the project, and the people are purely fictitious, but the situations and problems are real.

FIGURE 3.1

Change costs as a function of time in development

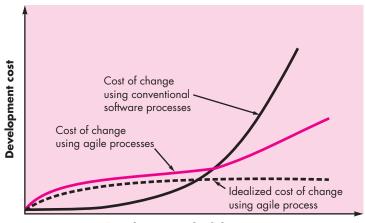


"Agility is dynamic, content specific, aggressively change embracing, and growth oriented."

–Steven Goldman et al.



An agile process reduces the cost of change because software is released in increments and change can be better controlled within an increment.



Development schedule progress

not adversely affect the outcome of the project. But what if we fast-forward a number of months? The team is in the middle of validation testing (something that occurs relatively late in the project), and an important stakeholder is requesting a major functional change. The change requires a modification to the architectural design of the software, the design and construction of three new components, modifications to another five components, the design of new tests, and so on. Costs escalate quickly, and the time and cost required to ensure that the change is made without unintended side effects is nontrivial.

Proponents of agility (e.g., [Bec00], [Amb04]) argue that a well-designed agile process "flattens" the cost of change curve (Figure 3.1, shaded, solid curve), allowing a software team to accommodate changes late in a software project without dramatic cost and time impact. You've already learned that the agile process encompasses incremental delivery. When incremental delivery is coupled with other agile practices such as continuous unit testing and pair programming (discussed later in this chapter), the cost of making a change is attenuated. Although debate about the degree to which the cost curve flattens is ongoing, there is evidence [Coc01a] to suggest that a significant reduction in the cost of change can be achieved.

3.3 What Is an Agile Process?

Any agile software process is characterized in a manner that addresses a number of key assumptions [Fow02] about the majority of software projects:

1. It is difficult to predict in advance which software requirements will persist and which will change. It is equally difficult to predict how customer priorities will change as the project proceeds.

WebRef

A comprehensive collection of articles on the agile process can be found at www.aanpo.org/articles/index.



- 2. For many types of software, design and construction are interleaved. That is, both activities should be performed in tandem so that design models are proven as they are created. It is difficult to predict how much design is necessary before construction is used to prove the design.
- **3.** Analysis, design, construction, and testing are not as predictable (from a planning point of view) as we might like.

Given these three assumptions, an important question arises: How do we create a process that can manage *unpredictability?* The answer, as I have already noted, lies in process adaptability (to rapidly changing project and technical conditions). An agile process, therefore, must be *adaptable*.

But continual adaptation without forward progress accomplishes little. Therefore, an agile software process must adapt *incrementally.* To accomplish incremental adaptation, an agile team requires customer feedback (so that the appropriate adaptations can be made). An effective catalyst for customer feedback is an operational prototype or a portion of an operational system. Hence, an *incremental development strategy* should be instituted. *Software increments* (executable prototypes or portions of an operational system) must be delivered in short time periods so that adaptation keeps pace with change (unpredictability). This iterative approach enables the customer to evaluate the software increment regularly, provide necessary feedback to the software team, and influence the process adaptations that are made to accommodate the feedback.

R POINT

Although agile processes embrace change, it is still important to examine the reasons for change.

3.3.1 Agility Principles

The Agile Alliance (see [Agi03], [Fow01]) defines 12 agility principles for those who want to achieve agility:

- 1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- **2.** Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- **3.** Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- **4.** Business people and developers must work together daily throughout the project.
- **5.** Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- **6.** The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- 7. Working software is the primary measure of progress.
- **8.** Agile processes promote s<mark>ustainable development.</mark> The sponsors, developers, and users should be able to maintain a constant pace indefinitely.



Working software is important, but don't forget that it must also exhibit a variety of quality attributes including reliability, usability, and maintainability.

- Continuous attention to technical excellence and good design enhances agility.
- **10.** Simplicity—the art of maximizing the amount of work not done—is essential.
- The best architectures, requirements, and designs emerge from selforganizing teams.
- **12.** At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Not every agile process model applies these 12 principles with equal weight, and some models choose to ignore (or at least downplay) the importance of one or more of the principles. However, the principles define an *agile spirit* that is maintained in each of the process models presented in this chapter.

3.3.2 The Politics of Agile Development

There is considerable debate (sometimes strident) about the benefits and applicability of agile software development as opposed to more conventional software engineering processes. Jim Highsmith [Hig02a] (facetiously) states the extremes when he characterizes the feeling of the pro-agility camp ("agilists"). "Traditional methodologists are a bunch of stick-in-the-muds who'd rather produce flawless documentation than a working system that meets business needs." As a counterpoint, he states (again, facetiously) the position of the traditional software engineering camp: "Lightweight, er, 'agile' methodologists are a bunch of glorified hackers who are going to be in for a heck of a surprise when they try to scale up their toys into enterprise-wide software."

Like all software technology arguments, this methodology debate risks degenerating into a religious war. If warfare breaks out, rational thought disappears and beliefs rather than facts guide decision making.

No one is against agility. The real question is: What is the best way to achieve it? As important, how do you build software that meets customers' needs today and exhibits the quality characteristics that will enable it to be extended and scaled to meet customers' needs over the long term?

There are no absolute answers to either of these questions. Even within the agile school itself, there are many proposed process models (Section 3.4), each with a subtly different approach to the agility problem. Within each model there is a set of "ideas" (agilists are loath to call them "work tasks") that represent a significant departure from traditional software engineering. And yet, many agile concepts are simply adaptations of good software engineering concepts. Bottom line: there is much that can be gained by considering the best of both schools and virtually nothing to be gained by denigrating either approach.

If you have further interest, see [Hig01], [Hig02a], and [DeM02] for an entertaining summary of other important technical and political issues.



You don't have to choose between agility and software engineering. Rather, define a software engineering approach that is agile.



"Agile methods derive much of their agility by relying on the tacit knowledge embodied in the team, rather than writing the knowledge down in plans."

Barry Boehm

What key traits must exist among the people on an effective software team?



"What counts as barely sufficient for one team is either overly sufficient or insufficient for another."

Alistair Cockburn

3.3.3 Human Factors

Proponents of agile software development take great pains to emphasize the importance of "people factors." As Cockburn and Highsmith [Coc01a] state, "Agile development focuses on the talents and skills of individuals, molding the process to specific people and teams." The key point in this statement is that *the process molds* to the needs of the people and team, not the other way around.²

If members of the software team are to drive the characteristics of the process that is applied to build software, a number of key traits must exist among the people on an agile team and the team itself:

Competence. In an agile development (as well as software engineering) context, "competence" encompasses innate talent, specific software-related skills, and overall knowledge of the process that the team has chosen to apply. Skill and knowledge of process can and should be taught to all people who serve as agile team members.

Common focus. Although members of the agile team may perform different tasks and bring different skills to the project, all should be focused on one goal—to deliver a working software increment to the customer within the time promised. To achieve this goal, the team will also focus on continual adaptations (small and large) that will make the process fit the needs of the team.

Collaboration. Software engineering (regardless of process) is about assessing, analyzing, and using information that is communicated to the software team; creating information that will help all stakeholders understand the work of the team; and building information (computer software and relevant databases) that provides business value for the customer. To accomplish these tasks, team members must collaborate—with one another and all other stakeholders.

Decision-making ability. Any good software team (including agile teams) must be allowed the freedom to control its own destiny. This implies that the team is given autonomy—decision-making authority for both technical and project issues.

Fuzzy problem-solving ability. Software managers must recognize that the agile team will continually have to deal with ambiguity and will continually be buffeted by change. In some cases, the team must accept the fact that the problem they are solving today may not be the problem that needs to be solved tomorrow. However, lessons learned from any problem-solving

² Successful software engineering organizations recognize this reality regardless of the process model they choose.

activity (including those that solve the wrong problem) may be of benefit to the team later in the project.

Mutual trust and respect. The agile team must become what DeMarco and Lister [DeM98] call a "jelled" team (Chapter 24). A jelled team exhibits the trust and respect that are necessary to make them "so strongly knit that the whole is greater than the sum of the parts." [DeM98]

Self-organization. In the context of agile development, self-organization implies three things: (1) the agile team organizes itself for the work to be done, (2) the team organizes the process to best accommodate its local environment, (3) the team organizes the work schedule to best achieve delivery of the software increment. Self-organization has a number of technical benefits, but more importantly, it serves to improve collaboration and boost team morale. In essence, the team serves as its own management. Ken Schwaber [Sch02] addresses these issues when he writes: "The team selects how much work it believes it can perform within the iteration, and the team commits to the work. Nothing demotivates a team as much as someone else making commitments for it. Nothing motivates a team as much as accepting the responsibility for fulfilling commitments that it made itself."



is in control of the work it performs. The team makes its own commitments and defines plans to achieve them.

3.4 Extreme Programming (XP)

In order to illustrate an agile process in a bit more detail, I'll provide you with an overview of *Extreme Programming* (XP), the most widely used approach to agile software development. Although early work on the ideas and methods associated with XP occurred during the late 1980s, the seminal work on the subject has been written by Kent Beck [Bec04a]. More recently, a variant of XP, called *Industrial XP* (IXP) has been proposed [Ker05]. IXP refines XP and targets the agile process specifically for use within large organizations.

3.4.1 XP Values

Beck [Bec04a] defines a set of five *values* that establish a foundation for all work performed as part of XP—communication, simplicity, feedback, courage, and respect. Each of these values is used as a driver for specific XP activities, actions, and tasks.

In order to achieve effective *communication* between software engineers and other stakeholders (e.g., to establish required features and functions for the software), XP emphasizes close, yet informal (verbal) collaboration between customers and developers, the establishment of effective metaphors³ for communicating important concepts, continuous feedback, and the avoidance of voluminous documentation as a communication medium.

³ In the XP context, a *metaphor* is "a story that everyone—customers, programmers, and managers—can tell about how the system works" [Bec04a].