Project Initiation Handbook

Laying the Foundation for Project Success

Karl E. Wiegers

PROCESS IMPACT

11491 SE 119TH DRIVE
CLACKAMAS, OR 97015-8778
PHONE: 503-698-9620 FAX: 503-698-9680
WWW.PROCESSIMPACT.COM

Contents

Introduction

Every software project manager knows of certain steps to take at the beginning of a project. You develop a business case, specify the product requirements, obtain funding and management sponsorship, assign a project manager, assemble the team, acquire other resources, and develop estimates and a project plan. I'm so confident that you know about these essentials from your project management training or previous experience that I'm not going to address them here.

However, several other activities are also vital to getting a software development project off to a good start. Unfortunately, project managers sometimes gloss over these steps. Perhaps they haven't had enough experience to realize how important these steps are, or maybe they don't feel they can spend the time during the frenzy of project launch. But seasoned managers know that paying attention to these critical activities can separate success from failure. These practices clarify project expectations and priorities, let stakeholders agree on what "done" means, and ensure continual improvement by learning from previous projects.

This handbook describes many actions that lay the foundation for a successful project. It doesn't attempt to cover every aspect of project initiation, instead focusing on these less-obvious activities. Both experienced and novice project managers will find these practices valuable. The focus is on software projects following any lifecycle or methodology (including agile), but the practices apply just as well to many nonsoftware projects. The handbook contains six chapters, each of which addresses a specific set of related practices for a key activity area:

1. Define Project Success Criteria
2. Define Product Vision and Project Scope
3. Define Product Release Criteria
4. Negotiate Achievable Commitments
5. Study Previous Lessons Learned
6. Conduct Project Retrospectives

Each chapter begins with a short scenario, describing an actual experience I have had. Each story illustrates a problem that can arise if a project manager neglects that chapter's principles and practices. The chapter then presents a tutorial that will let you put the chapter's topics into action. Common traps to avoid are identified with an icon of a mousetrap in the margin. True stories from real projects are flagged with a newspaper icon in the margin. Cross-references are provided to related chapters and to the relevant sections of the Project Management Institute's Body of Knowledge (PMBOK), the Software Engineering Institute's Capability Maturity Model® for Software (or just CMM®), and the CMM Integration® (CMMI®). Each chapter includes several practice activities. Worksheets are included so you can apply the practices to your own project immediately.

In my view, there is really no such thing as project management. What we call "project management" is a composite of managing many other project elements: people, communication, commitments, resources, requirements, change, risks, opportunities, expectations, technology, suppliers, and conflicts. Nearly every project includes aspects of all these activities and the successful project manager must keep an eye on them all. Use this handbook to help you start your next project on a solid foundation.

® CMM, CMMI and Capability Maturity Model are registered in the US Patent & Trademark Office by Carnegie Mellon University.

Chapter 1. Define Project Success Criteria

I've worked with some government agencies whose project funding is made available based on a state budgeting cycle. The project schedule is always presented as an imposed constraint: the project must *be done by the end of the budget cycle. But people in these agencies tell me that if they don't complete the project by that time, they nearly always get more money and time to finish the project in the next budget cycle. So the schedule really isn't a constraint at all, just a desirable target date.*

"Begin with the end in mind" is Habit 2 from Stephen Covey's *The 7 Habits of Highly Effective People* (Covey 1989). In Covey's words, "To begin with the end in mind means to start with a clear understanding of your destination. It means to know where you're going so that you better understand where you are now and so the steps you take are always in the right direction." At the beginning of every software project, the stakeholders need to reach a common understanding of how they will determine whether this project is successful.

Consultant Tim Lister defines *success* as "meeting the set of all requirements and constraints held as expectations by key stakeholders." If you don't know early on how you're going to measure your project's business success, you're headed for trouble. Defining explicit success criteria during the project's inception keeps stakeholders focused on shared objectives and establishes targets for evaluating progress. For initiatives that involve multiple subprojects, success criteria help align all subprojects with the big picture. Ill-defined, unrealistic, or poorly communicated success criteria can lead to disappointing business outcomes. Vague objectives such as "world class" or "killer app" are meaningless unless you can measure whether you've achieved them.

To help you determine where your project is heading, this chapter describes the four-step process for defining project success criteria shown in Figure 1-1 (Wiegers 2002d). The deliverables from this chapter will help you make the myriad decisions that pop up during the life of a project.

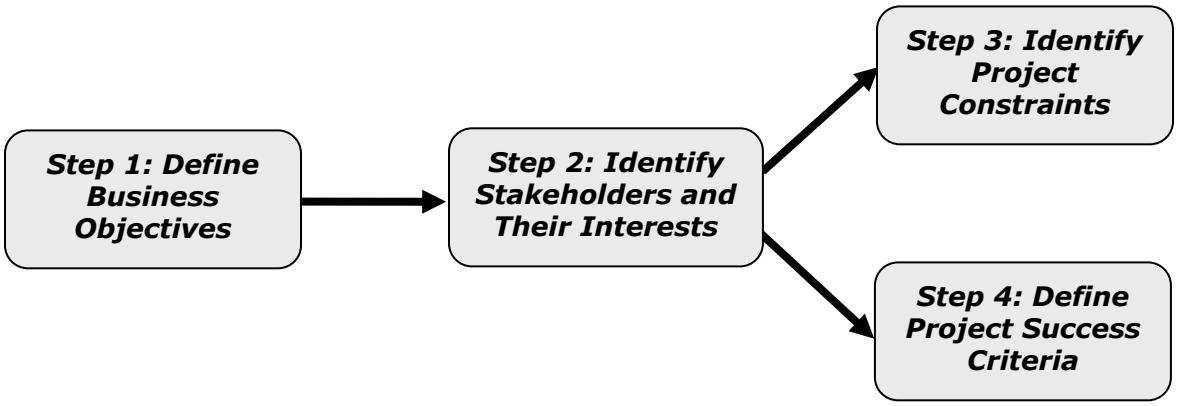


Figure 1-1: A process for defining success criteria

Step 1. Define Business Objectives

Business *goals* answer the question "Why do we want to tackle this project?" You can transform each goal into specific, attainable, verifiable, and prioritized business *objectives* to help you assess whether you've achieved the goal. Express business goals and objectives in terms of measurable business, political, or perhaps even social value. For example, a goal might be to "Demonstrate competitive advantage with distinguishing features and technologies." A corresponding business objective might state that specific features must be operational by a particular date. Another objective might require the team to demonstrate by a specific date that the planned technologies are sufficiently robust for use in your product (provided you can define what "robust" means).

A project that satisfies its requirements specification and ships on schedule and budget is good. A product that achieves its intended business objectives is much better. Well-crafted business objectives articulate the bottom-line outcomes the product will deliver for both internal and external stakeholders. They include the expected outcome, the time frame for achieving the objective, and how you will measure successful achievement (Wysocki and McGary 2003). Business objectives typically address:

- What the product[1] must be and do, including essential or distinguishing functions it will perform and how well it will perform them.
- Economic constraints, including development costs, cost of materials, and time to market.
- The product's operational and temporal context, such as technologies required for compatibility in the target environment, backward compatibility, and future extensibility.

Try to write business objectives that are SMART: specific (not vague), measurable (not qualitative), attainable (not impossible), relevant (not unimportant), and time-based (not open-ended). Table 1-1 illustrates some simple business objective statements.

Record your business goals and objectives in a high-level strategic guiding document for the project. Such documents include a vision and scope document (see Chapter 2), project charter, project overview, business plan, business case, or marketing requirements document. Some software project management plan templates include a section on management objectives and priorities, which could contain your business objectives and stakeholder analysis. A sample project management plan template is provided with this handbook. This template includes slots in which to store much of the information you'll develop if you complete the worksheets in this handbook. Other information might go into other project guiding documents, such as a quality assurance plan that contains product release criteria.

As the team gets into design and implementation, they might discover certain business objectives to be unattainable. The cost of materials might be higher than planned or cutting-edge technologies might not work as expected. Business realities also can change, along with evolving marketplace demands or reduced profit forecasts. Under such circumstances, you'll need to modify your business objectives and reassess to see whether the project is still worth pursuing.

One telecommunications project had an objective to build a replacement interface unit for an existing product at a specified maximum unit cost and with a defined reusability goal. A review of the proposed architecture revealed that the product would exceed the unit cost and couldn't meet the

[1] I'll use the terms "product," "application," and "system" to refer to whatever sort of software or software-containing deliverable your team is producing. These practices apply equally well to developing commercial software products, embedded systems, information systems, Web sites, and so on.

reusability goal. Management revisited the business case and concluded that the unit cost was still low enough to justify development. They revised the profit forecasts, dropped the reusability goal, and forged ahead. The key stakeholders redefined "success" to match business and technical realities.

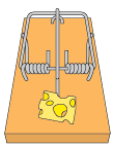
Table 1-1: Examples of Financial and Nonfinancial Business Objectives

Financial	*Nonfinancial*
• Capture a market share of X% within Y months. • Increase market share in region X by Y% in Z months. • Reach a sales volume of X units or revenue of $Y within Z months. • Achieve X% profit or return on investment within Y months. • Achieve positive cash flow on this product within Y months. • Save $X per year by replacing a high-maintenance legacy system. • Keep unit materials cost below X dollars per unit in the expected Y-year lifetime of the product. • Reduce support costs by X% within Y months. • Receive no more than X service calls per unit and Y warranty calls per unit within Z months after shipping.	• Achieve a customer satisfaction measure of at least X within Y months of release. • Process at least X transactions per day with at least Y% accuracy. • Achieve X time to market that provides clear business advantages. • Develop a robust platform for a family of related products. • Develop specific core technology competencies in the organization, with competency measured in some way. • Be rated as the top product for reliability in published product reviews by a specified date. • Maintain staff turnover below X% through the end of the project. • Comply with a specific set of Federal and state regulations. • Reduce turnaround time to X hours on Y% of customer support calls.

Step 2. Identify Stakeholders and Their Interests

A project achieves success by delivering suitable value to various *stakeholders*—people or groups that are actively involved in a project, are affected by its outcome, or can influence its outcome (PMI 2000; Smith 2000). Begin your quest for success by identifying these stakeholders and what is important to them. "Value" could translate to time savings for a corporate department, market dominance for a commercial software vendor, or increased productivity for a user. Look for stakeholders both inside and outside the development organization, in the categories shown in Table 1-2. Stakeholders can be involved in a project in many different ways (McManus 2005)

Next, perform a stakeholder analysis to reveal the expectations each stakeholder group has for the project. Identify each stakeholder's *interests*, or win conditions (Boehm and Ross 1989; Brackett 2001). Interests include financial benefits, specific time to market, required functionality, performance targets, usability, reliability, or other quality characteristics. Then evaluate how the pro-



TRAP: Overlooking important stakeholders, which means you're simply lucky if they view the project as a success.

Table 1-2: Some Internal and External Stakeholder Categories

Internal	*External*
• Project manager • Program manager • Product manager • Executive sponsor or funding authority • Project team members, including analysts, developers, testers, and technical writers • Quality assurance • Company owners or shareholders • Marketing • Manufacturing • Finance • Legal • Sales • Support	• Direct and indirect user classes • Procuring customers • Regulatory bodies • Auditors • Standards certification bodies • Government agencies • Subcontractors • Prime contractors • Venture capitalists • Business partners • Materials, information, and service suppliers

ject will be affected if each interest is or is not satisfied. The project might not be able to fully satisfy everyone's interests. This analysis will help you determine which interests are the most compelling. Figure 1-2 illustrates a simple template you can use to document essential stakeholder information.

Also assess the relative influence each stakeholder has on the project's decisions. Some stakeholders might wield great political power. Others could dictate essential features, impose restricting constraints, or be a potential source of substantial revenue. *Key stakeholders* are those who can strongly influence the project's decisions—and those whom it's most important to satisfy.

Expect to encounter conflicting stakeholder interests. Finance might want the lowest possible unit cost for the product, although the development team wishes to use expensive, cutting-edge technologies to meet stringent performance demands. Different market segments will value different combinations of time to market, feature richness, and reliability. Identifying your key stakeholders will help you to resolve such conflicts as they arise and to negotiate win-win solutions that maximize benefits for the greatest number of stakeholders.

Stakeholder	*Major Benefits*	*Attitudes*	*Win Conditions*	*Constraints*
Executives	increased revenue	see product as an avenue to 25% increase in market share within 1 year	richer and more novel feature set than competitors have	maximum budget = $1.4M
Manuscript Editors	fewer errors in their work	highly receptive, but unwilling to be retrained	automatic error correction capabilities; ease of use; high reliability	must run on existing low-end PCs

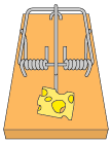
Figure 1-2: Simple stakeholder analysis example

Some years ago, my small software team was building an information system for a large corporation's research division. The stakeholder who represented our primary user group naturally wanted us to address just his department's needs. However, we identified other departments as also being important stakeholders. This led us to include functionality that made the system valuable to additional user classes. The primary stakeholder wasn't thrilled about the extra time it took us to deliver "his" application. But the stakeholder analysis helped us make the right strategic choices to best leverage the company's investment in this project.

Because of the many project decisions that lie ahead, it's essential to determine your key decision makers and to define a decision-making process very early in the project. Groups that must make decisions need to select an appropriate *decision rule* (Gottesdiener 2001). Some possible decision rules include:

- Voting, majority rules
- Reaching unanimous agreement
- Achieving consensus
- Delegating the decision to a single individual
- Having the person in charge make the decision after collecting input from others

There is no single correct decision rule, but every group needs to select one before they confront their first significant decision. Participative decision making generates more buy-in to the outcome than does unilateral decision making, which often is done by people with insufficient information about the problem being addressed.



TRAP: Failing to identify the most important stakeholders, thereby hampering effective decision making.

Step 3. Identify Project Constraints

Perhaps you've seen a sign in your local automobile repair shop that asks, "What do you want: good, fast, or cheap? Pick two." People often attempt to apply this classic "iron triangle" of tradeoffs to software. But I've seen the triangle drawn in many different ways, with different assumptions made about constant project scope or constant quality. Jeffrey Voas (2001) argues that it's not realistic to expect faster, better, *and* cheaper software. It is possible to build software faster and cheaper. But it's not possible to get software faster and better: You can't accelerate creativity, and software development is a highly creative activity. High-quality deliverables take a bit longer to initial release, but they save you much time in the long run because you don't have to do extensive testing and rework to fix and maintain them. Nor is cheaper and better feasible. Software with stringent quality demands, such as safety-critical systems, is expensive because it's imperative to remove (or, even better, prevent) defects.

The tradeoffs are real, but the triangle representation is wrong. In fact, a project manager must make tradeoffs along five dimensions: features (or scope), quality, staff, cost, and schedule (Wiegers 1996). Some people add an additional dimension, risk. Each dimension fits in one of three categories:

- *Constraints* impose boundaries and restrictions within which the team must operate. For constraints, state the immovable limit.

- *Drivers* identify key project success goals. Drivers typically afford the project manager a bit of latitude, but they define important targets toward which the team must work.
- *Degrees of freedom* are factors the project manager can adjust within certain limits. For degrees of freedom, identify the allowable range within which the project manager must operate.

It's important to classify the five project dimensions into these three categories early in your project. One way to represent this information is with a *flexibility diagram*, illustrated in Figure 1-3 (Wiegers 1996). A flexibility diagram is a Kiviat diagram (also known as a spider chart or radar chart), in which the five normalized axes extend radially from a center point. A point is plotted for each dimension on a scale from 0 (a constraint) to 10 (complete flexibility). Figure 1-3 illustrates a flexibility diagram for a hypothetical shrink-wrap software package. The point plotted at zero on the schedule axis indicates that schedule is a constraint. Staff and features are drivers with low amounts of flexibility. Cost and quality are degrees of freedom that offer more flexibility for the project manager. Connecting the points on all five axes creates an irregularly-shaped pentagon. Jim Highsmith describes an alternative "tradeoff matrix" to represent which project dimensions are fixed (constraints), flexible (drivers), or can be adjusted (degree of freedoms) (Highsmith 2004).

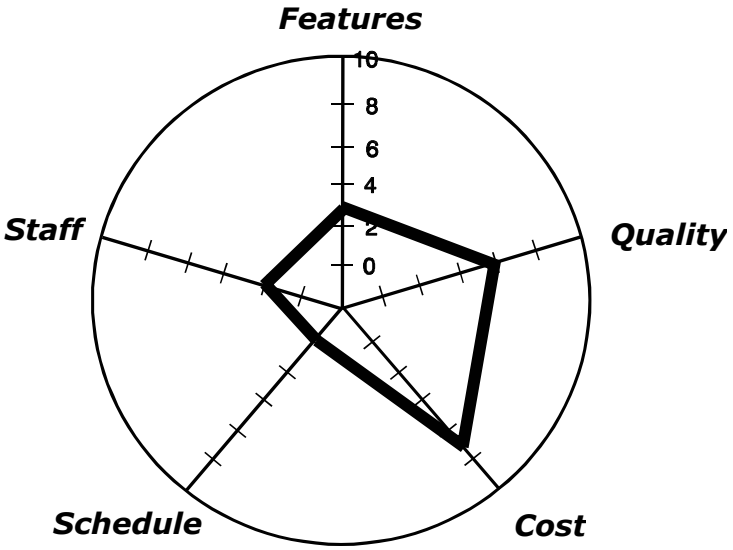


Figure 1-3: Flexibility diagram for a hypothetical commercial software product

The flexibility diagram is a qualitative tool intended to help the key stakeholders discuss project constraints and success drivers. Don't try to integrate the area of the pentagon or perform similar rigorous analyses. However, the smaller the pentagon, the more constrained the project is and the lower the chance of it being fully successful.

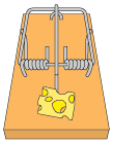
Managers often think of schedule as being a constraint when it's really a driver. To tell the difference, consider the consequences of shipping late. If it would mean paying a contractual penalty or losing a unique market window forever, then schedule truly is a constraint. One early e-commerce project delivered a set of Christmas-themed baskets bundled with a digital camera and special software. Had those baskets and the associated Web site not been available in time for Christmas, the project would have been a complete failure. Schedule really was a constraint in this case. Schedule is

a driver when there is a strong business reason to ship by a specific date but catastrophe doesn't strike if the release is delayed by a few weeks.

With an understanding of a project's drivers and constraints, the project manager can make appropriate tradeoff decisions. For example, maybe your project has sufficient funding but a hiring freeze is in place. Perhaps you can hire contractors, outsource some work, or buy some automation tools. Every tradeoff comes with a price tag, though:

- Adding more people is costly and doesn't shorten the schedule proportionately (Brooks 1995).
- Increasing scope can degrade quality if the schedule isn't extended.
- Stringent quality demands increase the development and testing effort needed.
- Attempting to reduce cost by skimping on requirements development or product design increases long-term costs and actually delays releasing an acceptable product.

There's more bad news: Not all project dimensions can be constraints. Project managers *must* have some latitude to react to schedule slips, scope growth, staff turnover, and other eventualities. A successful manager adjusts the degrees of freedom so as to let the team achieve the project's success criteria—its drivers—within the limits imposed by the constraints.



TRAP: Overconstrained projects that don't give the project manager enough flexibility to adjust to changing conditions.

Step 4. Derive Project Success Criteria

You can't judge whether a product satisfies all of its business objectives until some time after delivery. However, each business objective should imply technical success criteria you can monitor prior to release. The development team can't directly meet a business objective of "Capture a 40 percent market share within 12 months." However, they can decompose that objective into specific project actions and product features aligned with achieving the market share target. Disconnects between success criteria and business objectives give stakeholders no way to assess whether the project is likely to meet those objectives.

Success criteria shape many aspects of your project, beginning with the functional and nonfunctional requirements specifications. If the stakeholders understand the project's principal business objectives and success criteria, it's easier to make decisions about which proposed product features and characteristics are in scope and which are not.

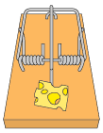
Table 1-3 shows some simple examples of project success criteria. Well-written success criteria are feasible, quantitative, and verifiable.[2] For example, performance goals are often written against either internal benchmarks or external industry reference data. A goal might compare a new search engine's performance to that of the prior version and also to the performance of a competing product under some set of standard conditions. Such success criteria let the project team design tests that measure whether performance, reliability, or throughput goals are being met. Trends in these

[2] Too often, success criteria are worded imprecisely and cannot be assessed objectively so it's difficult to judge if they've been satisfied. Tom Gilb has developed a flexible keyword notation called "Planguage" that permits precise statements of requirements and project goals (Gilb 2005). See Chapter 3 for a brief overview of Planguage and an example of how to use it to specify a product's release criteria.

Table 1-3: Examples of Project Success Criteria

- Total project cost does not exceed X% of the initial budget.
- The actual project duration is within X percent of the committed duration.
- All high-priority functionality defined in the requirements specification is delivered in the first release.
- The estimated number of residual defects does not exceed X per function point[3].
- Load testing confirms successful scale-up to X concurrent users, with Web page download times no longer than Y seconds over a 56K dial-up connection.
- All unauthorized network intrusions are detected, intercepted, blocked, logged, and reported.
- The mean time to failure under specified load conditions is at least X hours.
- The average installation time of the product is less than X minutes, with Y% of installations being error-free and requiring no support call.
- Prerelease development rework does not exceed X percent of total development effort.

measures may provide early warning that you might miss a success target, so the team can either take corrective action or redefine the success criteria.



TRAP: Unattainable, qualitative, or unverifiable success criteria. You'll never know if you're there.

Not all of these success criteria can be top priority. You'll have to make thoughtful tradeoff decisions to ensure that you satisfy your most important business priorities. Without clear priorities, team members can work at cross-purposes, which leads to rework, frustration, and stress. Weight your success criteria based on how strongly they align with achieving the critical business objectives, so team members will know which ones are essential and which are merely desirable. In one weighting scheme, stakeholders allocate 100 points to the success criteria associated with each business objective, with the more important items receiving more points.

Consider summarizing your success information in the form illustrated in Figure 1-4. List each business objective, the stakeholder who provided it, corresponding project success criteria, and each criterion's method of measurement and priority weight. Use the stakeholder list to verify that you haven't overlooked any important business objectives. You don't need to associate all stakeholders with every business objective, but every objective should be important to some stakeholder.

If you clearly define what success means at the beginning of your project, you greatly increase the chances of achieving it at the end. Understanding your stakeholders' interests, writing clear business objectives, and defining corresponding success criteria lay the foundation for a happy outcome.

[3] Function points are an implementation-independent measure of software size (IFPUG 2002).

Business Objective	Stakeholder	Project Success Criteria	Measurement	Weight
Achieve a customer satisfaction measure of at least 4 out of 5 within four months after release.	Marketing	Human factors approves user interface design through usability review.	All major severity UI design defects are corrected.	20
		UI prototype evaluation with focus group results in at least a 4.2/5.0 rating and will enable 90% of high-priority use cases to be performed.	Survey of focus group members; count of defined high-priority use cases.	30
		Product passes acceptance criteria defined by 80% of beta site evaluators.	Pass/fail rating by beta sites.	50

Figure 1-4: Sample Success Criteria Table

Cross-References

Define Product Vision and Project Scope (Chapter 2): The vision describes a solution that should satisfy the project business objectives. The scope defines the subset of that ultimate vision that the current project will address within the bounds imposed by project constraints.

Define Product Release Criteria (Chapter 3): Align your product's release criteria with the project's success criteria.

PMBOK 2.2: This section of the Project Management Body of Knowledge addresses project stakeholders (PMI 2000).

PMBOK 6.4.1.6: Two types of time constraints are discussed here: imposed dates, and key events or major milestones (PMI 2000).

Practice Activities

1. Complete Worksheet 1-1.
2. Complete Worksheet 1-2.
3. Select an appropriate decision rule and decision-making process that your stakeholders can follow to make strategic project decisions.
4. Complete Worksheet 1-3.
5. Complete Worksheet 1-4.
6. Identify stakeholders having conflicting success criteria. Based on your stakeholder analysis and your decision-making process, resolve these conflicts.
7. Based on the results of the above practice activities, think about what actions you might take if your project environment undergoes some significant change, such as those on Worksheet 1-5.

Worksheet 1-1: Your Project's Business Objectives

State several of your project's business objectives. Remember to make them SMART (specific, measurable, attainable, relevant, time-based).

ID	Business Objective
OB-1	
OB-2	
OB-3	
OB-4	
OB-5	
OB-6	
OB-7	
OB-8	

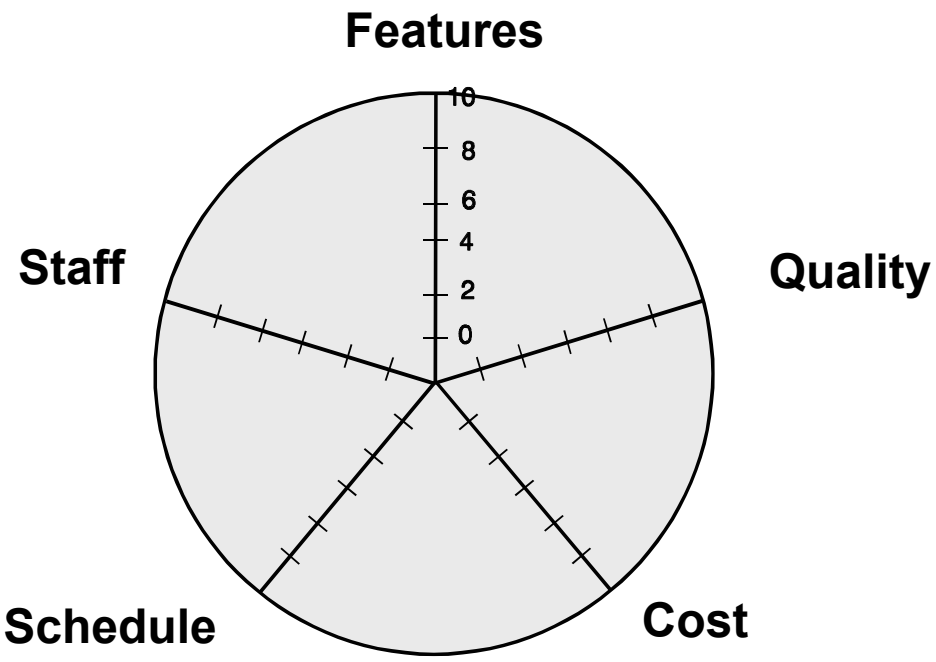
Worksheet 1-2: Your Project's Stakeholders

Identify and characterize your project's stakeholders. Determine which stakeholders carry more weight when resolving conflicting success criteria and priorities.

Stakeholder	Major Product Benefits	Attitudes	Win Conditions	Constraints

Worksheet 1-3: Your Project's Five Dimensions

Define your project's constraints, drivers, and degrees of freedom. Draw a flexibility diagram like that in Figure 1-3.



Dimension	Constraint (state limits)	Driver (state goals)	Degree of Freedom (state range)
Features			
Quality			
Cost			
Schedule			
Staff			

Worksheet 1-4: Your Project's Success Criteria

List your project success criteria, relating each to the stakeholders to whom it is important. To prioritize the success criteria within a business objective, distribute 100 points among them to indicate their relative importance. If you have multiple success criteria for each stakeholder and several key stakeholders for each business objective, use a separate worksheet 4 for each business objective.

Business Objective	Stakeholder	Project Success Criteria	Measurement	Weight
			Total	***100***

Worksheet 1-5: Responding to Change

Based on your business objectives, success criteria, constraints, and drivers, what actions might you take if the following events occur? Alternatively, identify incorrect assumptions or modified constraints that would require you to change the project's direction. How might you respond?

A competitor comes out with features you want to match?

Marketing moves the delivery date up 1 month?

Half the team quits to form a start-up?

New government regulations force changes in your requirements?

Your customer wants to add many new features?

Other changes specific to your project or application domain (describe)?

Chapter 2. Define Product Vision and Project Scope

A 30-person team was once building a product that provided point-of-sale and back-office services for a particular type of retail store. The project was more than a year behind schedule. One manager told me, "We blue-skied the requirements too much." By "blue-skying" she meant the team tried to include every suggested feature and function in the first release. They finally delivered the product, but uncontrolled scope growth nearly led to this company's demise before it shipped its first product. The team had a grand vision *for the ultimate product, but they didn't do a good job of defining a manageable* scope *for a series of planned releases.*

Chapter 1 described the importance of determining your project's business objectives. The product *vision* goes hand-in-hand with the business objectives. A vision statement provides a succinct description of the product you're creating and how it will satisfy the key stakeholders' objectives. The vision statement looks into the future and describes the long-term strategic target toward which the project is aiming.

However, rarely does the initial release of a new product fulfill the ultimate vision. Most products grow over time through a series of releases. Every project (or iteration) therefore must define its *scope*, a specified subset of the grand strategic vision that the current project will deliver. Defining the vision and scope helps you decide whether a proposed new feature should be included in the current product release, in a future release, or not at all. Some project teams write their vision and scope on a piece of poster board and bring it to their requirements workshops. This helps keep the workshop participants focused on both the long-term target and the near-term scope boundaries. The project's management sponsor is the owner and the keeper of the vision and scope definitions.

The Product Vision

Defining the product vision establishes a target or goal toward which the development team can focus its work. One study found that having a project manager with a clear vision of the project is a strong predictor of project success (Verner and Evanco 2005). The vision statement should be short, not a long rambling discourse that attempts to describe every product characteristic and how it will affect every potential stakeholder. Some people liken the vision statement to the "elevator test" (Highsmith 2004). How would you quickly describe your product to someone with whom you were chatting in an elevator for just a few moments?

Geoffrey Moore's classic treatise on marketing, *Crossing the Chasm*, contains an excellent keyword template for writing a vision statement (Moore 1991). Figure 2-1 shows a vision statement written using this template for one of my own company's products, an eLearning version of a training seminar. This vision statement identifies the target audience or market for the product, their principal needs, and the name of the product. It provides a brief description of the product's capabilities, characteristics, and benefits. It also contrasts the proposed product with the current reality, which

For	professional software practitioners
Who	wish to study requirements engineering but who cannot conveniently attend a live training session because of cost limitations, travel restrictions, time availability, or the number of students needed to justify an on-site class
The	"In Search of Excellent Requirements" (ISOER) eLearning CD
Is	a self-paced version of Process Impact's instructor-led ISOER seminar that combines animated Microsoft PowerPoint slides with audio recordings of Karl Wiegers narrating the slides. The CD will include links to HTML, PDF, Microsoft Word, and Microsoft Excel files that contain supporting materials such as magazine articles, a glossary of requirements engineering terms, document templates, sample requirements documents, spreadsheet tools, and other work aids.
Unlike	live seminar presentations, which are usually only 2 days in length and are expensive
Our Product	is not limited in duration or content. It will deliver high-quality training at a much lower per-student cost than does instructor-led training and will give the students much flexibility in how, when, and where they take the course. Selling the CDs will reduce the need for Process Impact instructors to travel. However, this product will not provide the opportunity for students to interact directly with the instructor.

Figure 2-1: A keyword-oriented sample vision statement

could be your existing product, a competitor's product, or (for a corporate information system) a current manual process.

You can see that this vision statement format provides a condensed, high-level understanding of both the product itself and the motivation for building it. It is certainly not a substitute for a full requirements specification. I find that writing a vision statement at the beginning of the project clarifies and crystallizes my thinking about the journey on which I'm embarking.

One of my clients, Bill, periodically asks me to develop a custom software engineering process, design a training course, or execute a similar project. I always ask Bill to write a vision statement for the work he's requesting. Bill does a bit of good-natured grumbling, but then he sends me a terrific vision statement in the format illustrated in Figure 2-1. This statement helps us share a clear understanding of where we're headed. I refer to the vision statement periodically to be sure my work is on track for achieving Bill's objectives and that I haven't overlooked anything.

In my training seminars I ask students to spend just five minutes writing a vision statement for their product. I'm always impressed by how much they get done in those five minutes. It's never too late to write a vision statement. If you don't already have one, I suggest you have several key stakeholders write their own vision statements independently, spending no more than 10 to 15 minutes on them. Then have the stakeholders compare their vision statements. Significant disconnects suggest fundamental differences of perspective or understanding about the product. You need to resolve these differences so the team can work in a coherent fashion toward a common objective.

The vision statement is the centerpiece of the vision and scope document, which is a high-level strategic guiding document for your project. As I mentioned in Chapter 1, other names for such strategic documents include a project charter, business case, or marketing requirements document.

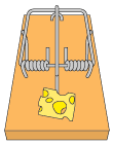
Figure 2-2 suggests a template for a vision and scope document, which is included with this handbook (Wiegers 2003). The template that describes what information to put into each section.

1. **Business Requirements**
   - 1.1 Background
   - 1.2 Business Opportunity
   - 1.3 Business Objectives and Success Criteria
   - 1.4 Customer or Market Needs
   - 1.5 Business Risks
2. **Vision of the Solution**
   - 2.1 Vision Statement
   - 2.2 Major Features
   - 2.3 Assumptions and Dependencies
3. **Scope and Limitations**
   - 3.1 Scope of Initial Release
   - 3.2 Scope of Subsequent Releases
   - 3.3 Limitations and Exclusions
4. **Business Context**
   - 4.1 Stakeholder Profiles
   - 4.2 Project Priorities
   - 4.3 Operating Environment

Figure 2-2: Template for vision and scope document

The vision and scope document defines your product's business requirements. This template includes categories for other strategic project information, such as the stakeholder profiles, business objectives, success criteria, and project priorities described in Chapter 1. You might prefer not to fully populate this template for a small or rapidly changing project. For any project, though, I strongly encourage you to look at each template section and consider if you should capture any information in that category to guide subsequent project activities.

The product's vision typically changes relatively slowly, although details will certainly evolve as development progresses and customers provide additional input and feedback. If you're planning an iterative approach or creating a series of incremental releases, as with an agile development project, the vision statement provides a thread of continuity. However, the scope of what each release intends to cover is more dynamic, responding to changes in desired functionality, technologies used, and business needs.



TRAP: An extended vision and scope document that's overkill for a small project. I once reviewed such a document with a vision statement a page and a half long. It described every project nuance from every stakeholder's perspective and it included a detailed technical description of the proposed product. Just one sentence in this document stated the central theme and project objective, so it could have been far shorter.

The Project Scope

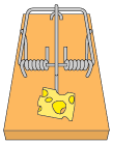
Many projects suffer from scope creep (a.k.a. feature creep or requirements creep), the ever-increasing demand for unexpected functionality that leads to delays, quality problems, and misdirected energy. Now, requirements will change and grow over the course of the project. This is a natural consequence of software development and you need to anticipate and plan for it. Scope creep, however, refers to the uncontrolled growth of features that the team attempts to stuff into an already-full project box. It doesn't all fit, and the continuing churn and expansion of the requirements make it difficult deliver the top-priority functionality as early as possible.

The first step in controlling scope creep is to document a clearly stated—and agreed to—scope for the project. Without such a scope definition, how do you even know you're experiencing scope creep? The second step for controlling scope creep is to ask "Is this in scope?" whenever someone proposes some additional product capability. If it's clearly in scope, the team needs to address the new functionality. If it's clearly out of scope, the team does not need to address it, at least not now. They might schedule the new capability for a later release.

Sometimes, though, the requested functionality is such a good idea that the current project scope should be expanded to accommodate it. This situation requires a negotiation between the project manager, the funding sponsor, and key customers to determine how best to handle the scope change. Thinking back to the five project dimensions from Chapter 1, you have the following choices, which you could combine:

1. Defer or eliminate some other functionality that was planned for the current release.
2. Obtain additional development staff to handle the additional work.
3. Obtain additional funding, perhaps to pay overtime (okay, this is just a little joke), outsource some work, or purchase productivity tools.
4. Adjust the schedule for the current release to accommodate the extra functionality (this is *not* a joke).
5. Compromise on quality by doing a hasty job that you'll need to repair later on.

Electing to increase project scope is a business decision that needs to consider cost, risk, schedule, and market implications. The people who are paying for the project need to make a considered decision as to which scope-management strategy is most appropriate in each situation. The objective is always to deliver the maximum customer value, aligned with the defined business objectives and project success criteria, within the existing constraints. Projects following an agile development lifecycle should write a scope statement for every iteration cycle to make sure everyone understands what will and will not be implemented during that iteration.



TRAP: Gold plating—adding unnecessary features that increase the project's cost but does not add proportionate value to the product. My team at Kodak tried to distinguish "steel" from "chrome." Our goal was to deliver the "steel" the customers required to get their jobs done, and to add as much of the requested "chrome" as we had time for.

There's no point in pretending the project team can implement an impossibly large quantity of functionality without paying a price. In addition, it's always prudent to anticipate a certain amount of scope growth over the course of the project. The wise project manager will incorporate contingency buffers into project plans so the team can accommodate some scope growth without demolish-

ing its schedule commitments (Wiegers 2002c). Sensible project scope definition requires that several conditions be met:

- The requirements must be prioritized, so the decision makers can select the capabilities to include in the next release. Prioritization requires input from both customers and developers. The key customer representatives, or "product champions," provide input regarding the benefit each proposed feature would provide and the penalty that could result if a feature was omitted or delayed. Developers assess the relative cost and technical risk associated with each proposed feature. A requirements prioritization spreadsheet that can help with this challenging task is included with this handbook (Wiegers 2003).
- The requirements must be sized so the team can judge how much effort it will take implement each requirement or feature. There is no perfect measure of software size. Some possibilities include: function points (IFPUG 2002); 3D function points (Whitmire 1995); counts of testable requirements (Wilson 1995); and the number and complexity of use cases or user stories (Jeffries, Anderson, and Hendrickson 2001).
- The team's productivity must be known, so you can estimate what quantity of functionality the team can realistically implement in an iteration of given duration. The agile methodology known as Extreme Programming terms this productivity measure *project velocity* (Jeffries, Anderson, and Hendrickson 2001).

The project scope establishes the boundary between what's in and what's out for a certain release. If stakeholders endlessly debate whether a specific feature is in scope or not, the scope likely is not clearly defined. Another sign of a vague scope boundary is that certain requirements are deemed in scope one week, then out of scope the next week, then back in scope again. The rest of this chapter describes three techniques for defining the scope boundary: a context diagram, a use case diagram, and feature levels.

Context Diagram

The *context diagram* is a venerable analysis model dating from the structured analysis revolution of the 1970s (DeMarco 1979; Wiegers 2003). Despite its age, the context diagram remains a useful way to depict the environment in which a software system exists. Figure 2-3 illustrates a partial context diagram for a hypothetical corporate cafeteria ordering system. The context diagram shows the name of the system or product of interest in a circle, which represents the system boundary. The context diagram reveals nothing about the internals of the system. The rectangles outside the circle represent *external entities*, also called *terminators*. External entities could be different user classes, organizations, other systems to which this one connects, or hardware devices that interface to the system. The interfaces between the system and these external entities are shown with the labeled arrows, called *flows*. Flows represent the movement of data, control signals, or physical objects between the system and the external entities.

The context diagram obviously shows the project scope only at a very high level of abstraction. This diagram tells us nothing about exactly what features are in scope. The features or functional behavior of the system are merely implied by the labeled flows that connect the system to the external entities.

Despite this limitation, the context diagram is a helpful representation of scope. I once taught a class on requirements for a corporate client. A student in the class showed me a context diagram she had drawn for her current project. She had just shown this to the project manager, who pointed out that a decision had been made to make one of the external entities—another information sys-

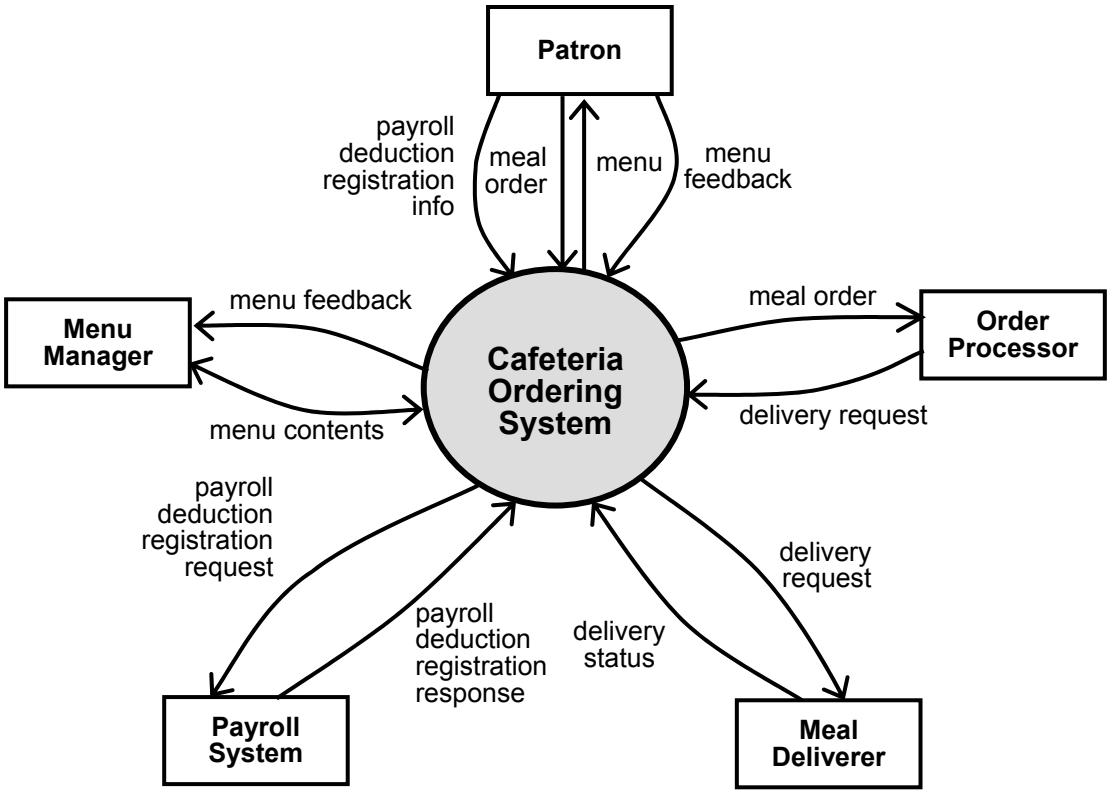


Figure 2-3: Sample context diagram

tem—part of the new system being developed. That is, the scope of the project just got larger. This student had expected that external system to be someone else's responsibility, but now it was her problem. The context diagram was a way to help the project stakeholders communicate a common understanding of what lies inside the system boundary.

Note that a context diagram could potentially represent either the ultimate vision for the final product or the scope for a specific project that will develop one release of that product. Both representations are appropriate, but be sure to label your context diagram so readers know exactly what they're looking at.

Use-Case Diagram

Use cases have become widely recognized as a powerful technique for exploring user requirements (Kulak and Guiney 2004; Wiegers 2003). Rather than focusing on product features during requirements exploration, the use case approach emphasizes understanding how users anticipate interacting with the product to accomplish business tasks or achieve useful goals. The Unified Modeling Language includes a *use-case diagram* notation (Armour and Miller 2001). See Figure 2-4 for a partial use-case diagram for our cafeteria ordering system.

The rectangular box represents the system boundary, analogous to the circle in a context diagram. The stick figures outside the box represent *actors*, entities that reside outside the system's con-

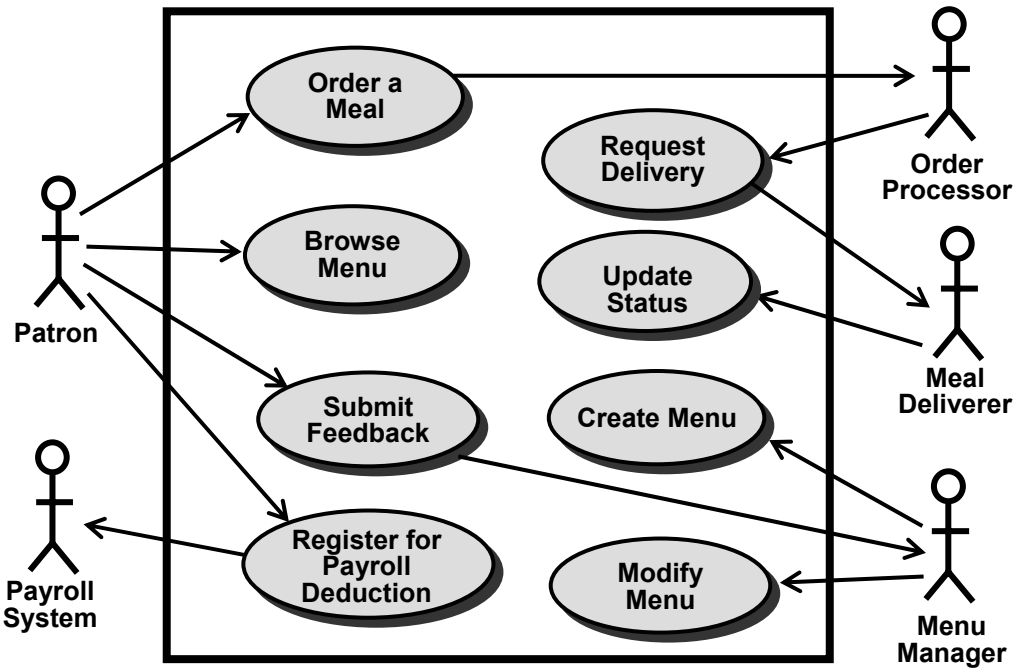


Figure 2-4: A sample use-case diagram

text but interact with the system in some way. These correspond approximately to the external entities shown in rectangles on the context diagram. Unlike the context diagram, the use case diagram does provide some visibility into the system internals. Each oval inside the system boundary represents an individual use case—literally, a case of usage—in which actors interact with the system to achieve a specific goal. The arrows indicate which actors participate in each use case. The use case diagram is slightly more useful for scope definition than the context diagram because you do get a first look at the contents of the system, not just its external interfaces.

Feature Levels

Each release of a product typically implements a certain set of new features—a group of logically related functional requirements—and perhaps enhances features that were partially implemented in earlier releases. We can think of each product feature as having a series of levels that represent increasing degrees of capability or feature enrichment (Nejmeh and Thomas 2002). That is, the development team adds new subfeatures or functional requirements when implementing the next release. One way to describe the scope of a particular product release, then, is to identify the specific level of each feature that the team will implement in that release. A sequence of releases represents increasing levels of functionality—and hence user value—delivered over a period of time.

To illustrate this approach to scope definition, consider the following set of features from our cafeteria ordering system:

FE-1: Create and modify cafeteria menus
FE-2: Order meals from the cafeteria menu to be picked up or delivered
FE-3: Order meals from local restaurants to be delivered
FE-4: Register for meal payment options
FE-5: Request meal delivery
FE-6: Establish, modify, and cancel meal service subscriptions
FE-7: Produce recipes and ingredient lists for custom meals from the cafeteria

Figure 2-5 indicates the various levels for these features that are planned for implementation in forthcoming releases. FE-2 and FE-5 in Figure 2-5 represent features with three enrichment levels each. The full functionality for each of these features is delivered incrementally across the three planned releases. The feature level approach (or "feature roadmap") is the most descriptive of these three techniques for defining the project scope.

Feature	*Release 1*	*Release 2*	*Release 3*
FE-1	Fully implemented		
FE-2	Standard individual meals from lunch menu only; delivery orders may be paid for only by payroll deduction	Accept orders for breakfasts and dinners, in addition to lunches; accept credit and debit card payments	Accept group meal orders for meetings and events
FE-3	Not implemented	Not implemented	Fully implemented
FE-4	Register for payroll deduction payments only	Register for credit card and debit card payments	
FE-5	Meals will be delivered only to company campus sites	Add delivery from cafeteria to selected off-site locations	Add delivery from restaurants to all current delivery locations
FE-6	Implemented if time permits	Fully implemented	
FE-7	Not implemented	Not implemented	Fully implemented

Figure 2-5: Sample feature description table

The farther into the future you look, the less certain the scope plans become and the more you can expect to adjust the scope as project and business realities change. Nonetheless, defining the product vision and project scope lays a solid foundation for the rest of the project work. This helps keep the team on track toward maximizing stakeholder satisfaction.

Cross-References

Define Project Success Criteria (Chapter 1): The product vision describes a solution that the stakeholders expect will satisfy the project's business objectives. The project success criteria help define what capabilities are in scope for each planned release.

Negotiate Achievable Commitments (Chapter 4): Effective scope management requires good-faith negotiation among project stakeholders to determine the functionality that each product increment or release will contain and the timing of those releases.

PMBOK 5: The Project Scope Management knowledge area involves initiating the project, planning and documenting scope of the project that will create the product, and subdividing the major project deliverables into smaller components. It also addresses obtaining formal acceptance of the project scope by the key stakeholders and developing a process for managing scope changes throughout the project (PMI 2000).

SW-CMM, Software Project Planning, Ability 1: This key practice states that "A documented and approved statement of work exists for the software project." The statement of work defines the scope of the project work to be performed (Paulk 1995).

CMMI, Project Planning, SP 1.1: This practice is titled "Estimate the Scope of the Project." The specific activities involve developing a work breakdown structure (WBS) to estimate the project scope. However, you can define the scope boundary for the project at a high level of abstraction without developing a WBS (Chrissis, Konrad, and Shrum 2003).

Practice Activities

1. Complete Worksheet 2-1.
2. Complete Worksheet 2-2.
3. Define the scope of your current project or iteration, using the use-case diagram (Worksheet 2-3), the feature description table (Worksheet 2-4), or both. Define any limitations or exclusions by stating what the product "is" and what it "is not."

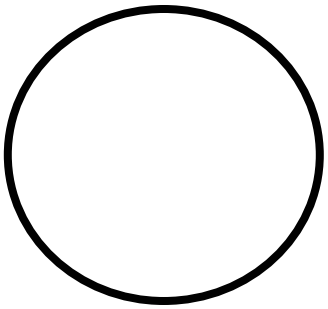
Worksheet 2-1: Product Vision

Write a vision statement for your product using the following template.

- ***For*** (target customer)
- ***Who*** (statement of the need or opportunity)
- ***The*** (product name)
- ***Is a*** (product category)
- ***That*** (key benefit, compelling reason to buy)
- ***Unlike*** (primary competitive alternative)
- ***Our product*** (statement of primary differentiation)

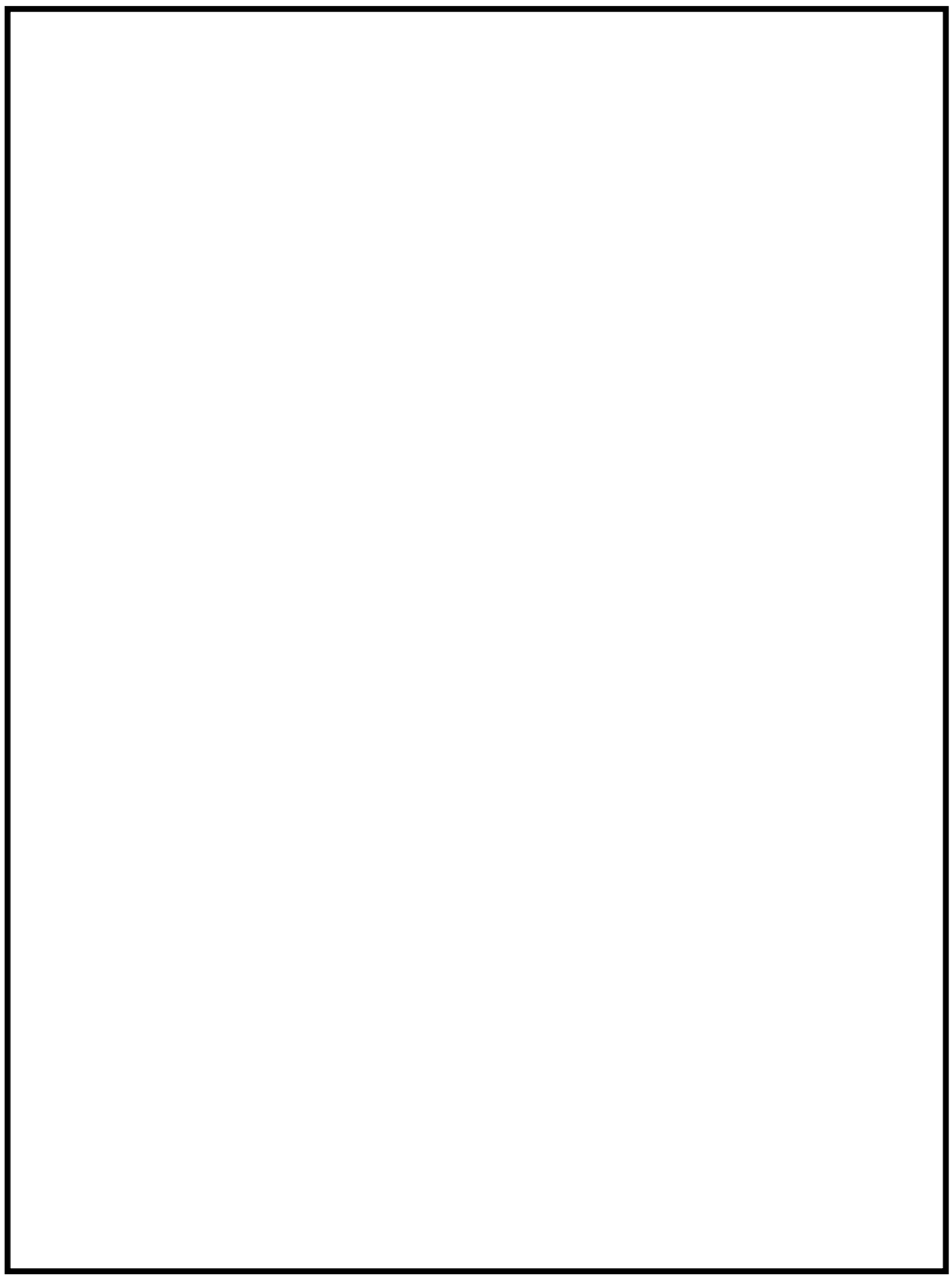
Worksheet 2-2: Context Diagram

Draw a context diagram for your current project or iteration.



Worksheet 2-3: Use-Case Diagram

Draw a use-case diagram for your current project or iteration.



Worksheet 2-4: Feature Description Table

Describe how you plan to implement and enrich product features over a series of releases.

Feature Descriptions:

FE-1: _____
FE-2: _____
FE-3: _____
FE-4: _____
FE-5: _____
FE-6: _____
FE-7: _____
FE-8: _____
FE-9: _____
FE-10: _____

Feature	Release 1	Release 2	Release 3
FE-1			
FE-2			
FE-3			
FE-4			
FE-5			
FE-6			
FE-7			
FE-8			
FE-9			
FE-10			

Chapter 3. Define Product Release Criteria

Several organizations I've encountered always meet their schedules for software projects. As the target date approaches and the project clearly won't finish on time, the team enters a "rapid descoping phase." They defer much planned functionality and release on the appointed completion date a crippled system with serious quality problems that provides no value to anyone. Then they declare the project to be a success because they delivered on time. Yes, they delivered something *on time, but that something bore a scant resemblance to the expected product.*

One aspect of beginning with the end in mind is to determine how to tell when you're ready to release the product. Defining your product's release criteria is one of the essential activities that lay the foundation for a successful project. "It's June 30 so we have to ship" isn't usually the best reason to conclude that you're done. Selecting the release criteria is the responsibility of the team's key stakeholders, with input from the rest of the development team. Whatever criteria you choose need to be realistic, objectively measurable, documented, and aligned with what "quality" and "success" mean to your customers. Decide early on how you will tell when you're done (and who will make the call), visibly track progress toward these goals, and make sure everyone understands the implications if they ship before the product is ready for prime time.

How Do You Know When You're Done?

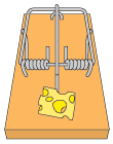
No one ever builds perfect software products that contain every imaginable function that behave flawlessly under all circumstances. Every project team needs to decide when its product is good enough to go out the door. Being "good enough" means the product has some acceptable blend of functionality, quality, timeliness, customer value, competitive positioning, and supporting infrastructure in place. You can always spend more time and money to make any product better, but at some point you need to draw the line. You need to balance the cost invested in the product against the value it will provide both to customers and to the organization building the software.

There is no perfect, simple measure of software quality. The customer view of quality depends on factors such as reliability (how long they can expect to use it without encountering a failure) and performance (response time for various operations). Strive to obtain customer input to the release criteria. How would your customers judge whether the product was ready to be installed and used, whether they could cut over to a new application that replaces a legacy app, or whether they'd be willing to pay for the new product?

The internal or engineering view of quality is also important. Is the software written in such a way that it can be modified and maintained efficiently during its operational life? Is documentation in place so the help desk or field support staff can assist users? The pro*ject* might end when you deliver the system, but the pro*duct* will live on for many years to come. The conditions that dictate releasability must consider the long-term implications.

Possible Release Criteria

There's no universally correct set of release criteria for all projects. Your release criteria must correlate with the project's success criteria (see Chapter 1). That is, whatever indicators you choose to tell you when the product is ready to ship must provide some confidence that the project will be an overall success. Because you must be able to measure and monitor the release criteria during development, you need to select surrogate indicators of the product's ultimate performance in the field. You might define separate release criteria for a series of releases that contain increasing functionality, reliability, or performance requirements.



TRAP: Weak release criteria that don't align with achieving the project's business objectives. These can give a false sense of confidence that the project will be a success.

By way of illustration, Table 3-1 lists release criteria for two products described in the public domain. One is the next release of a compiler and the second is a revised internetwork operating system that runs on a family of routers. These criteria reflect important ideas, but some of them are vague: "extensive customer exposure," "high level of customer satisfaction". Such fuzzy statements are subject to individual interpretation and can't be evaluated precisely to judge whether they're satisfied. Some ambiguous words to avoid are: acceptable, adequate, appropriate, comprehensive, extensive, high, improved, low, reasonable, robust, seamless, sufficient, and supported.

Table 3-1: Some Actual Release Criteria

Compiler	*Internetwork Operating System*
• Specific functionality is working correctly. • Specific platforms are supported. • Specific language front-ends are supported. • Specific existing regression tests all pass. • The compiler works correctly on specified real-world applications. • Specified performance benchmarks for code quality are achieved. • Compile speed is within a specified tolerance relative to benchmarks of previous compiler versions.	• The trend in the defect backlog is improving, based on defect density measurements and projected defects derived from software reliability engineering. • Release notes enclosures are complete. • The product has received extensive customer exposure. • There's a high level of customer satisfaction. • The product has undergone comprehensive system test and deployment in our internal network. • Customer Advocacy has evaluated known defects. • There are no open Critical Account Program issues. • There were no serious defects in previous maintenance releases. • No new features or platforms are addressed; defect resolution commits only, with minimal code changes.

Consultant Johanna Rothman recommends we write product release criteria that are SMART (Rothman 2002). This acronym reminds us that the criteria should be:

Specific (not vague or imprecise)

Measurable (not qualitative or subjective)

Attainable (not unrealistic stretch goals that you'll never achieve)

Relevant (related to customer needs and your business objectives)

Trackable (something we can monitor throughout the project)

These keywords are slightly different from the SMART criteria we encountered for business objectives in Chapter 1, but the basic idea is the same. Try applying these five criteria to the two examples shown in Table 3-1. You'll see that the release criteria for the internetwork operating system don't make the grade for several of the SMART conditions. I can anticipate energetic debates of whether the product has received enough customer exposure, what constitutes a "serious" defect, or whether the internal testing and deployment were sufficiently comprehensive. The criteria shown for the new compiler are easier to evaluate, provided the meaning of the word "supported" is made clear in every case. In general, I avoid using the word "support" in any kind of requirement—and release criteria define a type of requirement—because it's so ambiguous and context-dependent.

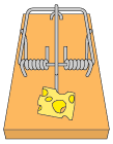
Table 3-2 shows a third example, release criteria for an open-source personal information management system called Chandler, developed by the Open Source Applications Foundation (http://wiki.osafoundation.org/bin/view/Chandler/ChandlerQAProcess). Most of these are clear enough that an objective observer could determine whether or not each was satisfied.

Table 3-2: Actual Release Criteria for a Personal Information Management System

Area	*Release Criteria for the Area*
Features	All features scheduled for the release are code complete and all feature development is frozen.
Quality Assurance	Functional and integration tests have run successfully [presumably thismeans the tests passed].
Certification	Testing building Chandler fully from the sources on all approved platforms, including software and hardware. Also testing the debug and end user's distributions on all the approved platforms.
Bugs	No blocker, critical or major bugs outstanding in Bugzilla [a defect-tracking system] for the target milestone set for this release.
Documentation	Full set of documentation for all the features delivered in the release. Also documentation of the features that were originally in the design but didn't make it in the release.
Performance Certification	For the performance criteria decided for the release, making sure the product performance conforms to that.

Consider the perspectives of various stakeholder groups to avoid disconnects and conflicts. User-developed acceptance criteria provide an essential input to the release criteria you select. Developers often don't really know what is important to the user, so work with key user representatives to make sure their perspectives are incorporated.

The following sections suggest some possible release criteria, grouped into several categories. Select conditions from these lists that will help you determine when it's time to take your product out of the oven and serve it. Of course, not all of these will apply to any single project and you might well have other criteria that are pertinent to your situation. Tailor these generic examples to suit your own project. Keep the SMART characteristics in mind as you write them, so your release criteria are as precise and meaningful as possible.



TRAP: Conflicting release criteria. All team members need to work toward a common set of goals and make the appropriate tradeoffs.

Defects

Everyone wants high-quality software, but the definition of "quality" has been debated for decades. Quality is a complex, multidimensional set of product characteristics. Releasing a buggy product prematurely can lead to high operational costs, user disappointment, poor product reviews, excessive maintenance costs, product returns, and even lawsuits. As one indicator of quality, you can monitor the number and types of defects discovered during development and testing. Of course, you don't know how many defects you *haven't* discovered yet. You only know how many you have found so far, how many you've fixed, and how fast the team is discovering new ones. It would be nice to estimate the frequency of likely software failure once the product is in the users' hands. However, the number of pre-release defects found is not a reliable predictor of the number of post-release failures to be expected (Sassenburg 2003).

You're never going to get perfect software, so you need some indicators that will suggest when the software is good enough to deliver. Trends in defect discovery rates can provide an indication of how many defects are likely to remain undetected in the product. The field of software reliability engineering provides some assistance. Various models have been developed to estimate the number of remaining defects from the rate at which new ones are being discovered through testing (Musa, Iannino, and Okumoto 1987; Walsh 1993). Some of the remaining defects will be inconsequential but others could be showstoppers. You just don't know.

If quality is a success driver for your project, consider adapting defect-related release criteria from the following general statements:

- There are no open defects with severity 1 or 2 on a 4-level severity scale.
- The number of open defects has decreased for X weeks and the estimated number of residual defects is acceptable.
- All errors and warnings reported by source code static analyzers and run-time analyzers have been corrected.
- Specific known problems from previous releases are corrected and no additional defects were introduced while making the corrections.

Testing

Most software teams rely heavily on testing of various types to discover defects, although other quality practices, such as inspections (Wiegers 2002a) and code analyzer tools, are also valuable. I'm not sure that most testers really know when they're done testing or can describe the state of

the product's quality when they declare testing complete. People sometimes stop testing because they're out of testing time. If you have historical data about the typical defect densities (defects per thousand lines of code, or KLOC) from previous projects you can estimate the number of defects likely to be hiding in your next product. You might decide to stop testing when the estimated number of undiscovered defects is acceptable, when testing has continued for a predefined number of hours with no failures, and when testing has continued for at least two weeks after reaching the specified estimate for remaining defects (Grady and Caswell 1987). Some other testing-related release criteria are:

- All code compiles, builds, and passes smoke tests on all target platforms.
- 100% of integration and system test cases were passed.
- Specific functionality has passed all system and user acceptance tests (for example, the normal flows and associated exception handlers for the most commonly executed use cases).
- Test case execution specified in the test plan is complete for all documented functional requirements.
- Predetermined testing coverage targets for code or requirements (e.g., functional requirements, use-case flows, or product features) have been achieved.

Combining consideration of testing-related and defect-related factors, one author proposes that your software is ready to ship when the following conditions are all satisfied (Foody 1995):

- You have a full regression test suite that covers 100% of the functionality and 80% of the branches.
- You have no known severity 1 or severity 2 defects.
- The density of known remaining defects is less than 0.5 defects per KLOC.
- The rate of arrival of new defects is less than 40 defects per 1000 hours of testing.
- The mean time between failures is at least 100 hours (that is, your interactive application runs continuously for at least 100 hours without something going wrong).
- You've completed stress, configuration, installation, localization, usability, and naïve-user testing.

These numbers might not be exactly appropriate for your situation, but the idea of quantifying several quality-related criteria provides valuable input into determining the right time to release. It's important to have multiple, complementary criteria. As an illustration, if no tests are executed in a certain period, obviously no defects will be found through testing. So the rate of defect discovery alone is not a dependable indicator for quality.

Quality Attributes

Another way to think about software quality is in terms of various attributes that characterize the product's behavior. Often called "the ilities" for short, these quality attributes include reliability, security, integrity, usability, portability, maintainability, efficiency, robustness, interoperability, and others (Wiegers 2003). The product's readiness for release can be determined partly based on whether it satisfies its critical specified quality attribute requirements. Here are some examples:

- Quantitative performance goals are satisfied on all platforms.
- Reliability goals are satisfied. (For example, the mean time between failures is at an acceptable level and is increasing with each build that goes through system test.)
- Specified security goals and requirements have been satisfied. (You might create a checklist of specific security issues and threats to examine. See http://rssr.jpl.nasa.gov/ssc2/questions.htm for an example of such a checklist.)

- Specified conditions have been met that will enable the product to pass a necessary certification audit or evaluation.

Functionality

Every product needs a minimal set of functionality that provides appropriate value to the customer. Careful requirements prioritization lets your team deliver a useful product as quickly as possible, deferring less urgent and less important requirements to later releases. Consider the following requirements-related release criteria:

- All high-priority requirements committed for this product release are implemented and functioning correctly.
- Specified customer acceptance criteria are satisfied.
- Any requirements for accessibility by disabled users are satisfied.
- All localization and globalization tests to ensure that the software runs properly in different languages were passed.
- Specified legal, contractual, standards-compliance, or regulatory goals are met.
- All functional requirements trace to test cases.

Configuration Management

The discipline of configuration management (CM) is akin to managing the parts list for your product. Configuration management includes activities to uniquely identify the components that are assembled into the product, to manage and control versions of these components, and to assemble (build) them properly into the deliverable files and documents. The following CM-related release criteria apply to virtually every software project:

- The product can be built reproducibly on all target platforms.
- A physical configuration audit confirmed that the correct versions of all components are present.
- The product installs correctly on all target platforms, including reinstall, upgrade from a previous release, uninstall, and recovery from aborted or failed install attempts.
- The release image and media have been scanned for viruses.

Support

You might think the software is done, but that doesn't mean the world is ready for it. Make sure you've lined up the other elements necessary for a smooth rollout and implementation. Think about factors such as these:

- Release notes have been prepared that identify defects corrected in this release, enhancements added, any capabilities removed, and all known uncorrected defects.
- Software release and support policies and procedures are published and understood by affected stakeholders.
- Known uncorrected defects have been logged in the project's defect-tracking system.
- The support function is ready to receive and respond to customer problem reports.
- The operating environment has all necessary infrastructure in place to execute the software.
- Software manufacturing and distribution channels are ready to receive the product. Manufacturing is especially important when the product includes software embedded into hardware components.

Precise Release Criteria with Planguage

As we saw with some of the examples in Table 3-1, release criteria are often written in a qualitative, subjective style that makes it difficult to know exactly what is intended and when you've satisfied them. To address this problem, consultant Tom Gilb has developed a notation that he calls "Planguage" (Gilb 2005). Planguage—derived from "planning language"—permits precise specification of requirements, project business objectives, and release criteria. Here's an example of how to express a release criterion using Planguage (Simmons 2001):

TAG: Reliability.Conversion

AMBITION: Very low frequency of failures when converting archive data files into current data formats.

SCALE: Percent of conversion failures when performing published supported file conversions.

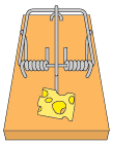
METER: Convert 400 files representing an operational profile of the supported archive data file formats. Express the number of failed conversions as a percentage of the attempted conversions.

MUST: No more than 5 percent.

PLAN: No more than 1 percent.

WISH: No more than 0.1 percent.

In Planguage, each release criterion receives a unique *tag* or label. The *ambition* describes what you are trying to achieve. *Scale* defines the units of measurement, and *meter* describes precisely how to make the measurements. You can specify several target values. The *must* criterion is the minimum acceptable achievement level for the item being defined. You haven't satisfied the release criteria unless all "must" conditions are completely met. The *plan* value is the nominal target and the *wish* value represents the ideal outcome. Expressing your product release criteria using Planguage provides a precise and explicit way to know whether your product is ready to go out the door.



TRAP: Unachievable, missing, or meaningless release criteria. None of these will help you steer the project toward a meaningful outcome. People don't take unattainable "stretch goals" seriously.

Marketplace Expectations

When choosing release criteria, keep in mind that different types of customers desire various combinations of time to market, features, quality, and value proposition. Geoffrey Moore divided the potential market for a product into the five customer categories shown in Figure 3-2 (Moore 1991). This analysis was derived from an earlier discussion about the diffusion of innovation by Rogers and Shoemaker (1971). Be sure to know your audience when you're trying to determine what "done" means for your product.

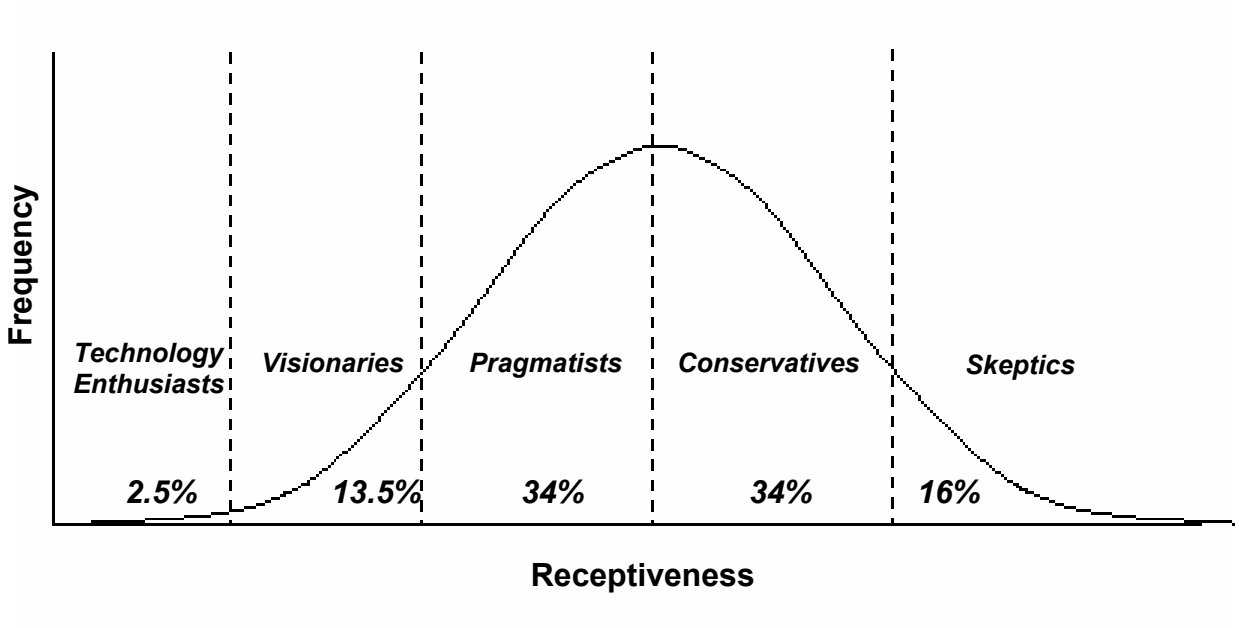


Figure 3-2: Distribution of market segments

Technology Enthusiasts are innovators. They get excited about brand new products they can get in their hands quickly. They're interested in cool new features and have a high tolerance for defects. They'll be eager to try out a beta version. However, they only represent a few percent of the potential market for a product. Just because your technology enthusiasts like the product doesn't mean that it's ready for the mass market yet.

Visionaries are the early adopters. They like new technology and new products. A visionary is willing to be the first one on his block to try out something different. Visionaries are opinion leaders in their organization. They like to explore fresh ideas but they're a bit more cautious than the technology enthusiasts. These are the kinds of people who are willing to wrestle with version 1.0 of a new product.

A large fraction of the market fits in the category of *Pragmatists*, also called the early majority. They adopt change more quickly than average but they don't have as much tolerance for quality problems as do the technology enthusiasts and the visionaries. If you're targeting the pragmatists, quality is more important than time to market, and feature set is generally even less of a success driver. Pragmatists don't want to work through the rough edges on the earliest releases.

About one-third of the potential market consists of the late majority, termed *Conservatives.* Conservatives don't adopt new methods and products until they're already well established in the marketplace. They wait for the product to mature so they can just get their work done with it, without having to cope with a lot of problems. Their priorities are reliability first, followed by a useful feature set. They're willing to wait for the product if necessary, so time to market isn't important to conservatives.

Skeptics make up the final fraction of the potential market. Also called laggards, they prefer to stick with their old ways of working and are reluctant to accept new ideas or tools. This is the "late kicking-and-screaming" market segment. You'll probably never reach them.

Making the Call

Regardless of what release criteria the team selects, you also need to know who will monitor progress toward satisfying the release criteria and who will make the final release decisions. I say "decisions" because you'll probably have several types of product releases. You might deliver a late-stage build to quality assurance as a release candidate for final system testing. After the product passes its QA checks, you might release it to a beta testing site for user acceptance testing, deliver it to manufacturing, or approve it for general availability release. Information systems being developed for internal corporate use typically go through a sequence of quality verification, beta test in a staging environment, and release to the production environment. Different stakeholders might make these various release decisions and different release criteria will pertain to each decision point.

As you select release criteria, think about how they can be measured and who will do the measuring. Keep these questions in mind:

- How, when, by whom, and to whom will the results of the measurements be reported?
- Who will evaluate and interpret the measurements?
- Who will make the ultimate decision that the criteria are satisfied and the product is ready for release?

Monitoring progress toward satisfying release criteria provides visibility into project status, showing whether the project is on track to meet its goals. Write and track release criteria in a binary fashion: each one is either completely satisfied or it is not. You might create a color-coded tracking chart for your release criteria:

- Green indicates that the criterion is completely fulfilled.
- Red means that the criterion is not yet fulfilled.
- Yellow indicates a risk of possibly not being able to completely satisfy the criterion.

Avoid using yellow as an indicator for partial satisfaction of the release criterion. The temptation would then be to base a release decision on how many yellow indicators you can tolerate. If the board isn't all green, you aren't ready to ship.

Both the project manager's and senior management's commitment to the release criteria are essential if the criteria are to mean anything. Make sure the project's senior management sponsor agrees that the release criteria are appropriate indicators of likely business success. Also obtain the sponsor's commitment to rely on these criteria to make the ultimate release decisions. If management elects to override the objective release criteria indicators, the team will question whether it's even worth the effort to devise the criteria and work toward them. Inappropriate decisions to release a product prematurely undermine the team's commitment to building high-quality products. They won't take release criteria seriously on future projects.

As the scheduled delivery date fast approaches, you might conclude that some release criteria won't be satisfied. To anticipate this possibility, establish a process for reevaluating those criteria to determine if it makes sense to modify them. Maybe marketing is concerned about missing the optimum marketplace time to release, but is it in the company's best interest to release an incomplete or deeply flawed product on schedule? The release criteria you select early on should reflect the appropriate balance of product features, quality, timeliness, and other factors that are aligned with business success. If you develop these criteria thoughtfully, you shouldn't have to change them unless business objectives or other key project realities change. Identify the risks of failing to satisfy each release criterion so the stakeholders understand the implications of ignoring or finessing them.

When I was a software engineer at a large corporation, customers used to approach me frequently with a request for some program they wanted me to write. Some of those people didn't want to spend time exploring their requirements; I was supposed to just dive in and get started. My question to those customers was, "If you don't have any requirements, how will we know when we're done?" None of them ever provided a satisfactory answer to this question, but this conversation made more agreeable to the idea of exploring requirements. "How will we know when we're done?" is a vital question to ask in the early stages of any project. Release criteria provide a valuable answer.

Cross-References

Define Project Success Criteria (Chapter 1): The product release criteria need to align with the project's business objectives and success criteria.

CMMI, Decision Analysis and Resolution. Determining when a product is ready to be released is a critical project decision. This process area addresses practices for analyzing potential decisions, identifying and evaluating alternatives, establishing a decision-making process, and selecting appropriate solutions from among the alternatives (Chrissis, Konrad, and Shrum 2003).

Practice Activities

1. Determine what market segments you're targeting with your next release, or with a series of planned releases. Think about the appropriate balance of time to market, value, functionality, quality (reliability, usability, and other attributes) that will be important for each market segment.
2. Complete Worksheet 3-1.
3. Complete Worksheet 3-2.

Worksheet 3-1: Your Product's Release Criteria

List several release criteria for your product. Make sure they are specific, measurable, attainable, relevant, and trackable. Determine what you can measure to see whether you're approaching satisfaction of that criterion. For those that can be quantified, state the target value that would indicate the criterion has been satisfied.

ID	Description	Measurement Method	Target Value	Implications If Not Satisfied

Worksheet 3-2: Decision Makers for Product Releases

List the project stakeholders who will make the product-release decision. There might be multiple release decisions, such as releasing a build to the quality assurance group, to beta test sites, to manufacturing, or to general availability. Describe the decision-making process each group will use.

Type of Release	Decision Makers	Decision-Making Process

Chapter 4. Negotiate Achievable Commitments

I once observed a discussion between a senior manager, Jack, and a project manager, Ron, about a new project. "How long will this project take?" Jack asked. "Two years," replied Ron. "No," said Jack, "that's too long. I need it in six months." So what did Ron say? "Okay." Now, what changed in those few seconds? Nothing! The project didn't shrink by a factor of four. The development team didn't get four times larger (although this would not have solved the problem). The team didn't become 400% as productive. The project manager simply said what he knew Jack wanted to hear. Not surprisingly, the project took more than two years to complete.

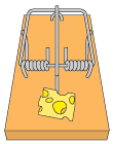
Too many software projects are launched with the hope that productivity magic will happen, despite previous experiences to the contrary. Sometimes a team does get lucky or pulls off a miracle. More often, though, miracles don't happen, leaving managers, customers, and developers frustrated and disappointed. Successful projects are based on realistic commitments, not fantasies and empty promises. This chapter presents several ways to improve your ability to make—and keep—achievable project commitments (Wiegers 2002b; Humphrey 1997).

Making Project Commitments

A commitment is a pact one person makes with another. Both parties expect the pact to be kept. The commitment might involve delivering a specific work product or performing a particular service by a specified time and with a certain level of quality. A software development project involves many commitments between customers, business managers, development managers, and individual developers. The project manager makes both individual commitments and commitments on behalf of the whole team. The project manager also requests commitments from team members and perhaps external stakeholders. Some software projects have at least one formal commitment: a legally binding contract.

Effective commitments must meet two conditions:

1. *We must make them freely.* Powerful people can pressure us to say yes, but most people are unlikely to take ownership of an impossible commitment that is thrust upon them. Some people will pretend to make a commitment they don't intend to fulfill to bring an unpleasant conversation to a close, but this undermines the commitment ethic.
2. *They must be stated explicitly and be clearly understood by all parties involved.* Ambiguity about the specifics of the commitment reduce the chance of it being satisfied.



TRAP: Imposed commitments. The commitments you strive to satisfy are those you make freely, not under duress.

It's easy to make unrealistic promises on software projects. Most developers are optimistic about their talents and overestimate their productivity. Project managers assume their staff have forty or more hours of productive project time available each week, although the reality is considerably less. Managers and customers who don't understand software development pressure the team to provide perfect estimates and to meet their aggressive business expectations with little regard for the uncertainties we face. Often, we get incomplete or misleading information about the problem we're asked to solve, which limits our ability to make meaningful estimates and commitments. Requirements are volatile, changing frequently for both technical and business reasons. We need the freedom to renegotiate unrealistic commitments that turn out to be based on inaccurate information or premises that have changed.

Negotiating Commitments

Negotiations are in order whenever there's a gap between the schedule or functionality that project stakeholders demand and your best prediction of the future as embodied in project estimates. Plan to engage in good-faith negotiations with customers, senior managers, and project team members about achievable commitments. *Principled negotiation*, a method to strive for mutually acceptable agreements, involves four key concepts (Fisher, Ury, and Patton 1991):

- Separate the people from the problem.
- Focus on interests, not positions.
- Invent options for mutual gain.
- Insist on using objective criteria.

Separate the People from the Problem. Negotiation is difficult when emotions get in the way. It's also hard to negotiate with individuals who wield more organizational power than you do. Before you dismiss a manager or customer as an unreasonable slave driver who makes impossible demands, recognize his legitimate concerns and objectives, including commitments others might have made that put *him* in a bind.

Focus on Interests, Not Positions. A product marketing manager who establishes a strong position in a negotiation ("This product absolutely must ship by November 5th!") can find it difficult to change that position. If you, the project manager, define an opposing but equally entrenched position ("We can't possibly be done before mid-January!"), conflict is inevitable. Rather than reaching an impasse by defending your terrain to the death, seek to understand the interests that underlie each party's stated position.

Perhaps marketing's real interest is to have a demo version available for the COMDEX trade show. Your interest might be to avoid burning out your team with massive overtime for four months and establishing a track record of failure. Maybe you and the marketing manager can agree on a core set of features and performance targets that would satisfy marketing's primary interest *and* be achievable by the convention deadline. Delivery of the completed product could then be deferred to a more reasonable date.

Invent Options for Mutual Gain. Negotiation is the way to close the gap between polarized positions. Begin with a win-win objective in mind (Boehm and Ross 1989), and look for creative ways to satisfy each party's interests. Think of feasible outcomes to which you are willing to commit and present those as options.

If you can't commit to fully satisfying a key stakeholder's demands, state what you *are* able to deliver. Could you make a more ambitious commitment if certain conditions were met, such as

shedding your team's maintenance responsibilities on existing products? If your team is pressured to work a lot of unpaid overtime to get the job done, will the company grant them compensating vacation time afterward?

Insist on Using Objective Criteria. Negotiations based on data and analysis are more constructive than those based on faith or opinions. Data from previous projects and estimates based on a rational planning process will help you negotiate with someone who insists that his grandmother could finish the project in half the time you've proposed. When you're relying on commitments from a third party, such as a subcontractor or vendor, trust data over promises. Keep these commitment tips in mind:

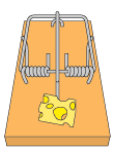
- When requesting commitments from other people, remember that an estimate is not the same as a promise. Expect to receive a range for an estimate, not a single number or date. Alternatively, request a confidence level, a probability that the estimate will be met.
- A common reason for commitment failure is making "best case" commitments rather than "expected case" commitments. Some teams define internal *target* delivery dates that are more optimistic than their publicly *committed* delivery dates. This sensible approach helps compensate for imperfect estimates and unanticipated eventualities.
- Improve your ability to meet commitments by creating more realistic estimates. Base your estimates on data of actual performance collected from previous activities.
- To avoid overlooking necessary work, use planning checklists that list all the activities you might have to perform for common large tasks.
- Contingency buffers provide protection against estimation errors, erroneous assumptions, potential risks that materialize, and scope growth.

Once some project requirements are defined, one approach to making commitments is to have team members "sign up" for specific task responsibilities (McConnell 1996; Jeffries, Anderson, and Hendrickson 2001). This is a characteristic of an organization with a healthy commitment ethic. Such voluntary personal commitments—based on each person's evaluation of what it will take to accomplish a goal—motivate team members to make the project succeed. A manager cannot impose this level of enthusiasm and commitment upon others.

Your committed delivery dates might be farther out than your managers or customers would like, but if you consistently fulfill the commitments you make, other people will know they can count on you and your team.

Documenting Commitments

When my brother Bruce was an engineering manager, periodically he would discuss expectations with individuals in his team. Sometimes Bruce would think a team member had made a commitment, but the team member viewed Bruce's request merely as a suggestion. To reduce ambiguity, it's helpful to write a brief summary of each commitment that you exchange with someone else. This confirms the communication and establishes a shared expectation of accountability. A written commitment record also permits tracking of actual performance against expectations. Unless I write them down, I find it easy to lose sight of both promises I've made to others and things I'm expecting to



TRAP: Verbal commitments that are unclear and undocumented. Negotiate to make interpersonal and intergroup commitments explicit and realistic. Renegotiate promptly if changes in circumstances won't permit you to keep them.

receive from other people. A simple table such as that in Figure 4-1 is a convenient way to document commitments.

Commitment	Made By	Made To	Due Date	Comments

Figure 4-1: Template for a commitment record

Modifying Commitments

Many projects have formal agreements to implement a specific set of requirements by a particular date. Such commitments can fall by the wayside if the requirements turn out to be technically unfeasible or especially challenging, if customers modify them, or if the stated requirements were just the tip of the *real* requirements iceberg. Project stakeholders must alter their expectations and commitments in the face of evolving information.

Inevitably, project realities—requirements, resources, technologies, budgets, schedules—will change. Problems will arise, risks will materialize, and new functionality will be requested. If your customers demand ten new features late in the development cycle, they can't expect to get the product by the original deadline. Whenever it appears that some targeted deliverables won't be completed before you run out of time and resources, the key project stakeholders need to decide how to respond. As described in Chapter 1, you need to remember which project dimensions are constraints, which are important success drivers, and which ones you can adjust if necessary. Here are some options you might explore, guided by your established project priorities:

- Can some lower-priority requirements wait for a later release?
- Can delivery be delayed? By how long?
- Can you channel more developer staff into the project?
- Will the customer pay extra to bring contractors on board to implement the new features?

If it becomes apparent that you or your team won't meet a commitment, inform the affected stakeholders promptly. Don't pretend you're on schedule until it's too late to make adjustments. Letting someone know early on that you can't fulfill a commitment builds credibility and respect for your integrity, even if the stakeholders aren't happy that the promise won't be kept.

Rigidly adhering to impossible or obsolete commitments can lead to serious problems. One project manager promised year-end delivery of a mission-critical application. He maintained that his small team could do the job without additional resources, but the year drew to a close without a delivery. The next year, his manager brought more team members on board and the project made good progress. Again, the project manager promised to deliver by the end of that year but again he failed to do so. Unwilling to admit that he couldn't achieve his commitment, he kept the challenges he was facing to himself, hoping to save face.

In fact, his stubbornness had the opposite effect. His development team lost credibility with both customers and managers. In addition, technology changes over the long course of the project rendered the new application largely irrelevant. When the system finally was delivered, it was nearly

dead on arrival and many stakeholders were disappointed. A more responsible manager would have acknowledged reality and renegotiated resources and delivery dates. And when it became clear that pursuing the initial requirements amounted to throwing good money after bad, the project decision makers should have redirected or canceled the project.

Your Personal Commitment Ethic

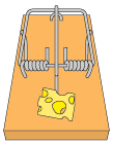
Well-intentioned people often commit to more than they can handle. I once managed a capable and dedicated developer who always said "yes" whenever I asked her to take on a new responsibility. I soon learned, however, that she often didn't deliver on schedule. Her in-basket was overflowing. There was simply no way she could complete everything to which she had agreed, despite her cooperative nature and good intentions.

Accepting more responsibilities than you can fulfill makes you look unresponsive and makes others skeptical about your promises. A meaningful commitment ethic includes the ability to say "no" (Karten 1994). Here are some other ways to say "no" that might sound more palatable:

- "Sure, I can do that by Friday. What would you like me to *not* do instead?" (As project manager, you're responsible for making these trade-off decisions.)
- "We can't get that feature into this release and still ship on schedule. Can it wait until the next release, or would you rather defer some other functionality?" (This is a way to say "not now" instead of "no.")
- "That sounds like something I can do, but I'm afraid it's not as high on the priority list as my other obligations. Let me suggest someone else who might be able to help you more quickly than I can."

A track record of making realistic performance estimates and satisfying your commitments earns credibility. That credibility improves your future ability to negotiate problematic schedules or other commitment requests. Before you say, "Sure, no problem" to a request, use the Commitment Analysis Checklist in Figure 4-2 to think through carefully whether you should make the commitment.

Despite occasional pressure to promise the impossible, I practice this personal philosophy: Never make a commitment you *know* you can't keep. My personal preference is to undercommit and overdeliver. Keeping that resolve is both part of being a professional and a way to be recognized as someone who is reliable. Once I was discussing process improvement plans with my department's aggressive and intimidating senior manager, four levels above me. Fred was pressuring me to agree to a timetable that my group had concluded was not remotely feasible. When I resisted, he grudgingly moved his objective out several months, but even that goal was pure fantasy. Finally I took a deep breath and said, "Fred, I'm not going to commit to that date." Fred literally did not know what to say. I don't think anyone had ever before refused to make a commitment he demanded.



TRAP: Unachievable commitments. People won't try as hard to achieve commitments they know are impossible. They don't want to set themselves up for failure.

Status	Commitment Criteria
❑	All parties fully understand what is being committed to.
❑	I have examined my assumptions and any external dependencies that could affect my ability to fulfill the commitment.
❑	I have identified circumstances that could prevent me from keeping the commitment. The likelihood that these circumstances will occur is: Not very likely ❑ Somewhat likely ❑ Very likely ❑
❑	The commitment is realistic and achievable, based on what I know today.
❑	I have documented the commitment, along with my assumptions and dependencies.
❑	I will notify the other parties if anything affects my ability to deliver on the commitment.

Figure 4-2: Commitment analysis checklist

It would be unprofessional to commit to an objective that I knew was unattainable, I explained. I said, "We're not going to work any less hard if we have more time to do it, and our morale will be higher if we're not set up for certain failure." Reluctantly, Fred agreed to a more plausible target date. He didn't shout at me or hit me or fire me, although in some situations those might be possibilities you need to consider.

As a project manager, don't pressure your team members to commit to requirements or delivery dates they don't believe to be achievable. If you fear someone is getting in over his head, discuss the commitment details, assumptions, the risks of failing to meet the commitment, and other obligations that could get in the way. Stay on top of the promises you and your team members make to others by recording and tracking requirements and milestones. Also track the promises others make to you. Create an environment in which your team members can turn to you for help with negotiation and priority adjustments. Unfulfilled promises ultimately lead to unhappy people and projects that don't succeed, so strive to build a realistic commitment ethic in your team.

Cross-References

SW-CMM, Software Project Planning, Activity 4: The essence of SW-CMM Level 2 is to negotiate, make, and keep realistic commitments. Senior management should review and approve commitments made to stakeholders external to the developing organization (Paulk 1995).

SW-CMM, Software Project Tracking and Oversight, Activities 3 and 4: Commitments need to be tracked, renegotiated commitments need to be reviewed, and project stakeholders need to be informed about commitment changes that affect them (Paulk 1995).

CMMI, Project Monitoring and Control, SP 1.2: Monitor significant documented commitments to determine whether they're at risk of not being satisfied (Chrissis, Konrad, and Shrum 2003).

Practice Activities

1. Complete Worksheet 4-1.
2. Identify commitments that others have made to you. Assess the impact if any are not satisfied.

Worksheet 4-1: Your Project's or Your Personal Commitments

List your personal commitments or commitments you've made on behalf of your team. Indicate whether they are realistically achievable, knowing what you know now. Identify your (or your team's) interests with respect to each commitment and the interests of the person to whom you've made the commitment. Are any of these commitments at risk of not being satisfied?

Commitment	Made By	Made To	Date Due	Achievable?	Your Interests	Their Interests

Chapter 5. Study Previous Lessons Learned

When I worked in the Kodak Research Laboratories, I had a friend named John who designed new black-and-white films. John began every new project by doing…nothing (apparently). I would find him sitting in his office, reading technical reports or research notebooks about similar projects or staring into space. John was learning, and thinking, and planning before he began performing expensive, time-consuming experiments. By taking the time to study previous experience, John was able to conduct well-designed experiments that efficiently led to high-quality products.

Software projects can derail in many ways. You can avoid making many classic mistakes if you study the software engineering and project management literature, as well as the lessons your own teams have learned on previous projects. The astute project manager will begin each new project by taking some time to study, reflect, and chart a safer course through the project management minefield. "Doing nothing" while examining the lessons of the past is a high-yield investment in the future (Boddie 1995). The overconfident project manager, in contrast, will rely solely on personal experience, memories, and the team members' intelligence and experience to weather any crisis and master all challenges. Hubris, arrogance, and cockiness aren't a solid foundation for project success.

Best Practices

People have been developing software for more than 50 years. During that half-century, a number of software engineering and management practices have emerged as being strong contributors to project success. These are known as *best practices*. It behooves all project managers and software practitioners to become familiar with the established set of best practices in their domain. Selectively applying appropriate industry and local best practices is a way to benefit from the lessons learned from thousands of previous projects.

Some people don't care for the term "best practices," protesting that it implies a global or absolute standard of excellence. Certainly, practices are context-dependent. Few techniques can be stated universally to be superior in every situation. In fact, I often use the term "good practices" instead of "best practices." Nonetheless, there are patterns of practices and techniques that consistently give better results than others. One set of criteria that's used to select best practices is the following (Fogle, Loulis, and Neuendorf 2001):

- Existence (the practice has actually been used; it's not just a theoretical concept)
- Importance (the evaluators of the candidate practice consider it to be significant)
- Effectiveness (the practice yields better results than alternative practices)
- Tangible benefit (an organization that uses this practice achieves meaningful benefits)
- Innovation (the practice might make use of innovative approaches, such as automation of previously manual steps)
- High value perceived by users (the practitioners feel the practice works well for them)

Many best practices are collected by industry gurus and consultants who examine many projects for patterns that correlate with success and failure. One such group was called the Airlie Council. In the mid-1990s, the Airlie Council identified nine key best practices for software engineering and software management (Brown 1996). A later report expanded this to 16 best practices (Brown 1999). Six of these 16 practices apply directly to software project management:

- Adopt a program risk management process.
- Estimate empirically cost and schedule.
- Use metrics to manage.
- Track earned value.
- Track defects against quality targets.
- Treat people as the most important resource.

Other collections of best practices come from authors who glean and synthesize insights from the vast body of software engineering literature. An excellent example is Steve McConnell's fine book *Rapid Development*, which describes 35 best practices (McConnell 1996). McConnell presents the potential benefits from each practice, the effects on project schedule and risk, improvement in progress visibility, and the chances of both initial and long-term success. He also describes how to apply each best practice and the risks associated with each one. Nearly one-third of these 35 practices pertain to project management:

Lifecycle Practices

- Evolutionary delivery (a lifecycle model that delivers small increments of functionality at frequent intervals)
- Lifecycle model selection (choosing an appropriate sequence of activities for your project)
- Spiral lifecycle model (an iterative lifecycle model that emphasizes reducing risk early in the project)
- Staged delivery (an iterative lifecycle model that delivers a product through multiple partial releases)
- Timebox development (a strategy of adjusting scope to achieve a useful deliverable on a fixed schedule)

Planning and Tracking Practices

- Measurement (collecting and analyzing project data to assist with tracking, decision making, and future planning)
- Miniature milestones (decomposing a body of work into small pieces that can be estimated and tracked easily)
- Principled negotiation (a negotiation strategy that focuses on achieving win-win outcomes)
- Theory-W management (a project management approach that strives to achieve all stakeholders' win conditions)
- Top 10 risks list (a dynamic list of the most significant risks that pose a threat to project success)

One aspect of continuous improvement is to thoughtfully apply practices that many other projects have found to be valuable, provided they're appropriate for your project and team culture. However, it's also a good idea to recognize the classic mistakes project managers make so you can avoid them. The lessons painfully learned because something didn't go as planned are the most vivid. You don't have time to make every mistake that every previous project manager has already made, so you need to learn from their experience and suffering. *Rapid Development* identifies many classic mistakes, including the following (McConnell 1996):

- Adding people to a late project (Brooks 1995).
- Unrealistic expectations.
- Lack of effective project sponsorship
- Lack of stakeholder buy-in.
- Wishful thinking.
- Overly optimistic schedules.
- Insufficient risk management.
- Contractor failure.
- Insufficient planning.
- Abandonment of planning under pressure.
- Insufficient management controls.
- Omitting necessary tasks from estimates.
- Planning to catch up later.
- Push-me, pull-me negotiation.
- Overestimated savings from new tools or methods

A term related to "classic mistake" is *antipattern*. An antipattern "describes a commonly occurring solution to a problem that generates decidedly negative consequences" (Brown et al. 1998). As with patterns, which describe broadly applicable solutions to recurrent problems, antipatterns have concise and sometimes whimsical names. The antipatterns describe: the general form that the undesirable practice takes; its symptoms, consequences, and causes; and a better solution. Project management antipatterns include:

- Analysis Paralysis (thrashing on requirements and models in a quest for perfection early in the project)
- Death by Planning (spending excessive effort on multiple levels of highly detailed plans, which are assumed to define a perfect path to a successful project)
- Irrational Management (the manager cannot make decisions, direct development staff, or deal with a crisis)
- Project Mismanagement (key project activities, such as quality control steps, are overlooked or minimized, which leads to excessive defects found during integration and system testing)
- Smoke and Mirrors (customers are led to believe that impossible capabilities can be delivered)

Becoming familiar with software industry best practices, classic mistakes, and antipatterns can help every project manager choose and apply the most appropriate set of practices for a given situation. As you gain project management experience, you'll accumulate your own body of techniques on which to rely and traps to avoid.

Lessons Learned

In addition to published reports of best practices, we all have our own collections of lessons learned from our personal experiences or those of our colleagues. Cancelled projects or those that struggled to deliver a product provide a rich source of insights that can save your team grief on the next project, if you take the time to glean and apply those lessons. The most effective way to reap this harvest of knowledge is through a project retrospective (see Chapter 6). During a retrospective, project participants reflect on the experiences of the previous project, phase, or iteration. Some of these reflections will lead to lessons regarding different approaches to try the next time around. Successful projects are even more valuable, identifying effective solutions to common problems and suggesting actions to repeat on future projects.

Don't expect all project managers and team members to read the complete report from every retrospective and draw their own conclusions. For the maximum organizational leverage, accumulate and share lessons learned from your retrospectives, process assessments, and risk-management experiences. Organize the lessons-learned repository to make it easy for future project managers to find what they're looking for. The time you spend accumulating lessons and studying that collection will be amply repaid on every project that finds effective shortcuts and avoids repeating past problems.

I like to write lessons in a neutral way, so it isn't obvious whether we learned each one because we did something well or because we made a mistake. The template in Figure 5-1 provides a structure for collecting lessons from multiple projects within your organization. Accumulating insights from multiple projects in a structure like this (or, even better, in a database) facilitates sharing these lessons across all projects and teams.

ID	Date	Project	Description	Activity	Contact	Recommendations

Figure 5-1: Lessons learned template

The information in your lessons-learned repository should include:

- A unique identifier for each lesson. This could be a simple sequence number, but a more meaningful label or tag is better.
- The date the lesson was entered into the repository.
- The name of the project that contributed this lesson.
- Description of the lesson learned, including the risks of ignoring the lesson.
- The project life cycle activity, work product, or other subject category to which this lesson applies. Some possible choices are: Concept Proposal, Requirements, Architecture, Prototyping, Hardware, Detailed Design, Construction, Unit Testing, Integration, System Testing, Acceptance Testing, Certification, Installation, Maintenance, Process, Planning, Estimation, Tracking, Management.
- The name of a person the reader could contact to learn more about this lesson.
- Suggestions for implementing the lesson on a future project or for taking process improvement actions based on the lesson.

You can always add more information to the repository, such as searchable keywords, but this will get you started. NASA has made a public lessons-learned database available at http://llis.nasa.gov/llis/plls/. There you can see a richer style for documenting lessons learned and read some actual lessons from the space program.

Here are a few simple examples of lessons you might learn on a project. Grow your own list of lessons and pass them along to anyone who might benefit from your painful experience.

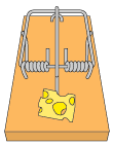
- Inspectors need to be trained before you can expect them to be highly effective at finding defects and to participate constructively in a positive inspection experience.
- It will take twice as long for new team members to get up to speed and become fully productive as you think it will.

- Requirements specifications that are going to be outsourced for implementation must be far more detailed and clearly written than those that will be implemented locally because there are fewer opportunities for informal interactions to resolve ambiguities and answer questions.

Cross-References

Conduct Project Retrospectives (Chapter 6): A retrospective provides an excellent opportunity to learn from your local experiences previous projects or from earlier iterations or phases in the current project. If you don't conduct retrospectives, project managers must rely only on their personal experience, reading, general software best practices, and training courses.

PMBOK 4.3.3.3, PMBOK 5.5.3.3, PMBOK 6.5.3.3, PMBOK 7.4.3.6: Your historical database of previous project information and insights should include the causes of size, schedule, cost, and effort variances, as well as both effective and ineffective corrective actions that were tried (PMI 2000). This information will assist future project managers who take the time to study it when planning their own projects.



TRAP: Assuming that unexpected occurrences on a previous project won't happen again. Unless you correct the underlying cause of a problem, it's likely to recur.

Practice Activities

1. Complete Worksheet 5-1.
2. Complete Worksheet 5-2 to plan how you wish to begin your approach to better project management.

Worksheet 5-1: Lessons Learned

Study your two most recently completed (or canceled) projects and identify lessons from them that might save time on your next project.

ID	Date	Project	Description	Activity	Contact	Recommendations

Worksheet 5-2: Getting Started with Improved Project Management Practices

Select several project management practices that you wish to learn more about and to apply to your work. Consider three time frames: next week (and I mean that literally), in about a month, and in about six months.

	Next Week	Next Month	In 6 Months
New practices to try			
Situation you might apply them to			
Benefits you hope to gain			
Help or additional information you might need			
Whose cooperation you might need			
Barriers that might prevent you from succeeding			
Who could break down those barriers			

Chapter 6. Conduct Project Retrospectives

My friend Hank worked on a project that failed. The organization decided to try again. Hank suggested that they conduct a project retrospective to determine why the first project failed and feed that insight into the next attempt. "No," the project manager replied, "we don't have time to hold a retrospective. We need to get started on the new project right away." My interpretation of this response is, "We need to get started immediately making the same mistakes again because it will take us so long to recover from them."

Retrospectives provide a structured opportunity to look back at the project, phase, or iteration you just completed (Kerth 2001; Wiegers and Rothman 2001). You might identify practices that worked well that so you can repeat them. You can also learn what didn't go well so you can make changes. This might not seem like part of project initiation—indeed, it's an important part of project close-out—but you can use the insights gained from a recent retrospective on the next project you undertake. You can't afford to repeat past mistakes and encounter the same surprises on project after project.

Retrospective Defined

A retrospective is a gathering of knowledge, insights, metrics, and artifacts from a completed project or project phase, which helps the team learn how to do a better job on future projects. Metrics might include estimated and actual results for schedule, effort, cost, and quality, which are essential to improve your future estimates. You can archive key project artifacts—documents, plans, specifications—in your process assets library for other projects to study or reuse. As Figure 6-1 illustrates, the outputs from the retrospective also feed into your organization's lessons learned collection, a list of

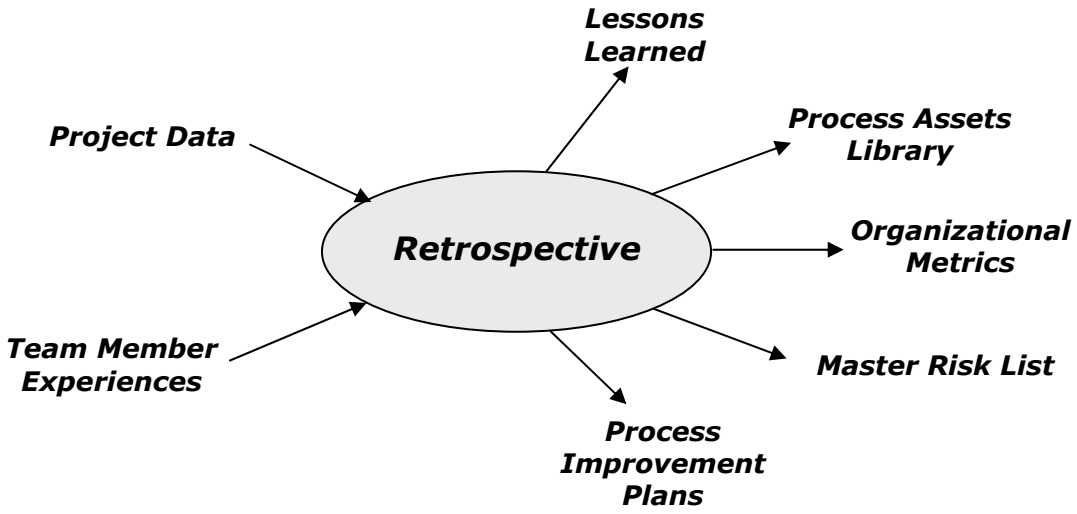


Figure 6-1: Inputs and outputs for a retrospective

potential risks, and future software process improvement activities.

A retrospective provides closure, a way for the participants to share their observations away from the day-to-day work pressures. Even if the project was a colossal failure, the lessons you learn from evaluating it can produce something positive from the experience and point to useful improvement opportunities (May 1998).

This wrap-up activity goes by various names. Some people refer to it as a "postmortem" (even if the project was successful) or an "autopsy" (when it wasn't). Other synonyms include "debriefing," "after-action report," "quality improvement review," and "post-project review." Consultant Norm Kerth, author of the definitive book on this topic, advocates "retrospective" as a neutral term that suggests a contemplative reflection on previous experience to gain practical wisdom (Kerth 2001).

You don't need to wait until the project is over to learn from it. Consider holding a retrospective whenever you want to gather information about your project or evaluate how the work is going. Many projects pass through a series of development iterations, so you should gather lessons after each iteration to help with future iterations. Reflecting on past experience is especially worthwhile any time something went particularly well or particularly badly. It's easy to forget what happened early in a project that lasts longer than a few months, so hold a short retrospective after reaching each major milestone on a long project. The healing effects of time and the exhilaration of recent success can ease the pain you suffered some time ago, but in those painful memories lie the seeds of future improvements.

The Retrospective Process

An effective retrospective follows the process depicted in Figure 6-2. The key players are the manager who sponsors the retrospective, the project team members, and a facilitator. A detailed procedure for planning and performing a retrospective is included with this handbook.

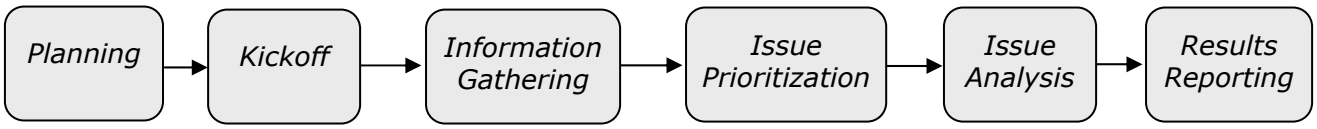


Figure 6-2: A retrospective process

Planning

Planning begins by including the retrospective as a set of tasks in your project schedule. The management sponsor who requested the retrospective works with the facilitator to determine the scope of the retrospective (the entire project or just a portion), the project activities to examine, and specific issues to probe. Select retrospective participants from the list of project stakeholders; see Table 1-2 in Chapter 1 for some possible internal and external stakeholder categories. Choose an appropriate facility for the retrospective, preferably off-site and away from distractions. Also, select the

facilitation techniques you will use and create an agenda. Kerth (2001) recommends a number of activities or "exercises" to perform during the retrospective.

Ideally, you'll be able to assemble all project participants in the same room for a constructive and honest retrospective. Sometimes, though, the presence of certain individuals or groups might skew the outcome. In one retrospective, another facilitator and I led two separate discussion groups in parallel. One group consisted of six managers, while 15 software practitioners comprised the second group. The facilitators and the sponsoring manager believed that practitioners would shy away from certain issues if their managers were in the room. Splitting the participants worked well in this situation, although we had to merge and prioritize the issues from both groups.

In another case, though, separating the participants proved unwarranted. The two groups involved were a software development team and a visual design team who had partnered on a major Web site development project. I underestimated the participants' collaborative mindset. Despite some points of friction between the groups, it would have been better to keep the entire team in the same room to discuss their joint issues.

It's important to prepare the managers by telling them that the retrospective may uncover issues that are uncomfortable for them. As Jerry Weinberg points out, "No matter how it looks at first, it's always a people problem," (Weinberg 1985) and sometimes the people problems are management-generated. Consider requesting permission from the managers to ask them to leave the room if they seem to be inhibiting the discussion.

Kickoff

During a short kickoff session with all participants, the sponsoring manager introduces the facilitator and any other unfamiliar faces. The manager also identifies his objectives for the retrospective and thanks the participants in advance for their time and their honest contributions. The manager should clearly state his commitment to taking concrete actions based on the retrospective outcomes. Without this commitment and the actual follow-through, team members will quickly lose any motivation to participate in retrospectives. The facilitator then outlines the agenda of events. To establish the appropriate constructive environment, the facilitator defines some ground rules. All participants must accept the ground rules and agree to abide by them. Typical rules include:

- Allow everyone to be heard.
- Respect the other person's experience and perspective.
- Think about root causes, not just symptoms.
- Avoid criticizing input and ideas from other participants.
- Avoid blaming people for past events.
- Focus on understanding, learning, and looking ahead.

Information Gathering

Some retrospectives collect hard data and project artifacts that hold significance for individual participants. Another valuable activity is to develop a time line of significant project events. A good way to do this is to ask participants to place sticky notes describing key events on a long sheet of butcher paper that represents the time span of the project (Kerth 2001).

The core retrospective activity is gathering issues, observations, and concerns from the participants. You might begin with a survey to collect input from the team on a wide variety of project

topics and issues (Collier, DeMarco, and Fearey 1996). Use the retrospective to explore four basic questions about the project:

1. What went well? (We want to repeat it in the future.)
2. What could have gone better? (We might want to change it.)
3. What happened that surprised us? (These might be risks to watch for on future projects.)
4. What do we still not understand? (These areas need further investigation.)

An experienced facilitator has a whole bag of tricks for eliciting information from participant groups. A traditional approach is to have a facilitator stand by a flipchart in front of the group and ask participants to suggest issues. The facilitator marks things that went well with plus signs and less favorable experiences with minus signs. In the round-robin variation, each participant contributes one issue in turn. The facilitator cycles through the group and until everyone passes on making further comments. If you use one of these approaches, plan to spend one to two hours on issue-generation. The facilitator or a scribe records each issue on a separate index card or sticky note. Then the participants group the cards into related categories (affinity groups) and name the groups.

These traditional facilitation methods have several drawbacks.

- Sequential issue-generation is slow.
- It's easy for a few outspoken participants to dominate the input.
- It's easy to slip into an extended discussion of a single hot-button topic, instead of identifying additional issues.
- Some participants may be uncomfortable raising issues in a public forum.
- Influential or strong-willed participants might inhibit others.

If you're concerned about any of these factors, consider using alternative facilitation approaches. Silent techniques that let participants generate issues in parallel can be more efficient and comprehensive than the public, sequential method. Begin by asking the participants to contemplate their own issues silently for a few minutes. Then have small groups of participants capture the issues and ideas on index cards, sticky notes, or flipcharts. Finally, reconvene the whole group to review, react, and analyze.

In another silent, parallel approach, the facilitator and retrospective sponsor identify several categories in which issues are likely to arise prior to the retrospective meeting. Common categories include: communication, organization, teamwork, management, requirements, design, construction, testing, subcontractors, business issues, and processes. You probably won't need all of these categories for every retrospective.

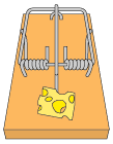
Next, write each category name on a separate flipchart page, and divide each page into labeled sections for what went well, what could have gone better, and what lessons were learned. During the meeting, have the participants write each of their issues on a separate sticky note, indicating the pertinent category. The facilitator places these in the appropriate section of the appropriate flipchart page. Spend about 20 minutes clarifying the things that went well, then move on to what could have gone better for another 20 or 30 minutes. Participants can walk around the room and see what other people wrote on the flipcharts to stimulate their own thinking.

This approach addresses most of the shortcomings of the traditional facilitator-at-the-flipchart method. Participants working concurrently can generate more issues in a given amount of time. The group won't become distracted by discussions as each issue is raised. And people who might be reluctant to state their opinions aloud willingly contribute them silently and anonymously. However, the facilitator will have to read all the sticky notes on the flipcharts aloud to share them with the entire group, make sure each issue is clearly stated and properly classified, and group related or dupli-

cate issues. There is a also risk that defining categories in advance will limit what comes out of the group, so you might prefer to group issues and name the groups following issue generation.

To close the data-gathering portion of a retrospective, consider asking each team member to answer two questions:

1. What one aspect of this project would you want to keep the same on a future project?
2. What one aspect of this project would you want to change on a future project?



TRAP: The retrospective becomes more about politics or blaming than about introspection and improvement. This will give the participants a bad attitude toward future retrospectives and can damage the team culture.

Issue Prioritization

A successful retrospective will generate more issues than the team can realistically address. You need to identify those items that the participants agree would be the most valuable to pursue. Some high-priority issues might point to effective practices you want to institutionalize on future projects. Others will reflect shortcomings in current practices that you need to address promptly.

One prioritization technique is Pareto voting. Each participant gets a limited number of votes, about 20 percent of the total number of issues being prioritized. Colored adhesive dots work well for this voting process. The participants walk around the room, examine the flipcharts, and place their dots on those sticky notes with the issues they believe are most important. The issues that gather the most dots are most ripe for early action. However, seeing the dots on the sticky notes can bias participants who might not want to "waste" votes on issues that clearly are not favored by the earlier voters. To sidestep that problem, you could have the participants place their voting dots on the backs of the sticky notes. Or, have everyone first study the flipcharts and select the issues for which they wish to vote, then have them all place their voting dots concurrently after making their selections.

Issue Analysis

If you have time during the retrospective, spend about 15 minutes discussing each high-priority issue revealed during data-gathering. Otherwise, assemble a small group to explore those topics after the retrospective meeting. Table 6-1 suggests some actions to take for the four different types of observations described on page 62. Some issues might fall outside the sphere of influence of the team members. The facilitator will need to bring those to the attention of the appropriate managers or stakeholders to be addressed.

Results Reporting

Share the retrospective results with all participants, any project stakeholders who did not participate in the retrospective, and senior managers in the project manager's reporting chain. You might need to escalate some issues to senior management. Senior managers who monitor progress on action plans coming out of the retrospective can strongly motivate the project manager and team to imple-

ment those plans (Whitten 1995). A template for a retrospective report is included with the retrospective procedure.

Table 6-1: Possible Actions for Various Retrospective Observations

Observation Category	*Suggested Action*
Something that went well	Determine why it succeeded and what benefits it provided. Find ways to ensure that those aspects of the project will go well again in the future.
Something that could have gone better	Determine why the item didn't turn out as intended, the consequences, and recommendations for doing it better the next time.
Something that surprised us	Add a new item to the organization's master list of potential project risks to consider.
Something we still don't understand	Determine how you can understand the issue, perhaps through research or root cause analysis. See Wiegers (2003) for an example of root cause analysis.

Retrospective Success Factors

A retrospective can succeed only in a neutral, non-accusatory environment. Honest and open communication is essential. If a project has been difficult or unsuccessful some venting is to be expected. However, the facilitator must limit that venting and channel it in a constructive direction. Make sure your retrospectives don't turn into witch hunts. The retrospective must emphasize guilt-free learning from the shared project experience. Consider the following critical success factors.

Define your objectives. The sponsoring manager should identify the retrospective objectives and the specific project aspects on which it should focus. Also consider the potential consequences of the activity. Who will come out ahead if the information gathered during the retrospective guides some constructive process changes? Who might look bad if the root causes of problems are revealed? You're not looking for scapegoats, but you need to understand what really happened and why.

Use a skilled and impartial facilitator. It isn't realistic to expect the project manager to facilitate his own project's retrospective objectively. The manager might have a particular axe to grind, want to protect his own reputation, or put his own spin on certain issues. Some project managers might unconsciously impede participation despite their good intentions. Other participants can be intimidated into silence on sensitive points.

To avoid these problems, invite an experienced, neutral facilitator from outside the project team to lead the retrospective. The facilitator's prime objective is to make the retrospective succeed by surfacing the critical issues in a constructive environment. Consider having someone who is not an active retrospective participant act as scribe to record the issues generated so the facilitator can concentrate on, well, facilitating.

Engage the right participants. Of course, the essential participants are all of the project team members and other key stakeholders. Managers are invited to the retrospective only if they actually worked with the project team. However, you should provide a summary of lessons learned to senior management or to other managers in the company who could benefit from the information.

Some teams might be too busy, too large, or too geographically separated for all team members to participate in a retrospective concurrently. In such a case, select representatives of the various functional areas that were involved in the project. If a large project was divided into multiple subprojects, each one should perform its own retrospective. Delegates from each subproject can then participate in a higher-level retrospective at the overall project level.

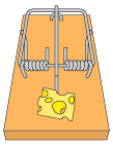
If the project involved multiple groups who blame each other for the project's problems or who refuse to sit down together to explore their common issues, you might begin by discussing the friction points between the groups. Chances are good that you'll uncover important project issues. If the groups can't get along in the retrospective, they probably clashed during the project, too. The retrospective might address what needs to change for those groups to collaborate more effectively the next time.

Prepare the participants. An invitation to a retrospective can stimulate fear, confusion or resistance if the participants aren't accustomed to retrospectives or if the project had serious problems. Some participants might be sick with anxiety, while others will be eager to let the accusations fly. It's important to build trust and establish a constructive mindset. Provide information and reassurance to the participants in the invitation material and through "sales calls" made on team leaders and key participants. Make sure everyone understands the retrospective objectives and the activities they'll be contributing to. Describe the process in advance. Emphasize that this is a future-oriented and process-improvement activity, not a blame fest.

Focus on the facts. A retrospective should address the processes and outcomes of the project, not the participants' personalities or mistakes. The facilitator has to concentrate on what actually happened and ensure the participants don't blame or placate others. However, people often perceive events in different ways. Understanding their different interpretations can release hard feelings and provide the opportunity for new insights.

Protect privacy. Identify the information coming out of the retrospective that can be shared with the rest of the organization and with senior management. Some personal issues and conflicts might be revealed during the retrospective. In general, though, it's not important who raised each issue, so it's important to protect the privacy of those who contribute their thoughts and recommendations.

Identify the action plan owner. Identify the person who will write and take ownership of an improvement action plan and see that it leads to tangible benefits. This owner must carry enough weight in the organization to steer the people who execute the action plan toward completing their action items.



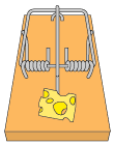
TRAP: Recommendations from the retrospective that are not specific, actionable, or realistic. Don't expect such recommendations to do future projects any good.

Action Planning

After the retrospective, don't try to tackle all the issues identified immediately. Choose up to three issues initially from the top of the priority list; the rest will still be there for future follow-up. Write an action plan that describes your improvement goals, identifies steps to address them, states

who will take responsibility for each activity and lists any deliverables that will be created. Assign each action item in the plan to an individual who will implement it and report progress to the action plan owner. At your next retrospective, check whether these actions resulted in the desired outcomes. An action plan template is included with this handbook.

An action plan that doesn't lead to concrete actions is useless. I once facilitated two retrospectives for the same development group two years apart. Some issues that came up in the later event were the same as those identified 24 months earlier. Failing to learn from the past practically guarantees that you will repeat it.



TRAP: An action plan is never created, or it is created but never implemented. If this happens more than once, the team won't believe management is serious about making changes.

The project team is the best source of information about how a recently completed project really went. You can use a retrospective to help your team assemble a rich picture of the project, so the manager can use that information to create a more effective environment the next time. But all organizational change takes time, patience, and commitment from all stakeholders. If people really don't want to change, they won't.

Cross-References

Study Previous Lessons Learned (Chapter 5): For the maximum organizational benefits, accumulate lessons learned from your retrospectives in a form from which future project managers can benefit.

Practice Activities

1. Identify points in your project life cycle at which performing a retrospective would be valuable.
2. Select a recently completed project on which to perform a retrospective, one whose experiences and learnings would assist you in planning and executing your next project. Use Worksheet 6-1 to develop a plan for this retrospective. Refer to the retrospective procedure to help you complete this worksheet.
3. Hold the retrospective and add lessons learned to your organization's lessons learned collection.

Worksheet 6-1: Retrospective Planning

Project: _____ **Planning Date:** _____

Item	Considerations	Your Plan
Sponsor	Who is the management sponsor for the retrospective?	
Objectives	What does the sponsor want to accomplish as a result of the retrospective?	
Beneficiaries	Who are the target beneficiaries of the retrospective?	
Scope	Single project or multiple projects? Exactly what is being reviewed? What functional areas are included? Are specific issues of particular interest for exploration?	Project(s): Functional Areas: Issues:
Participants	Who will participate in the retrospective?	
Deliverables	What documentation will result from the retrospective?	Interim Documentation: Final Documentation:
Issue Generation	Will issue generation take place prior or during the review meeting? Using what methods?	Prior or During: Methods:
Number of Meetings	How many meetings should be planned, based on the scope of the retrospective?	
Agenda	List the retrospective meeting components and the time allocated to each.	Introduction: Timeline: Gather Issues: Cluster: Prioritize: Choose Key Issues: Root Cause: Recommend Improvements:

Techniques	What methods will be used for the major retrospective activities?	Cluster: Prioritize: Choose Key Issues: Root Cause: Recommend Improvements:
Roles	Who is the facilitator? Who is the scribe? What are their responsibilities?	Facilitator: Responsibilities: Scribe: Responsibilities:
Management Role	What is management's role in the context of the retrospective?	
Metrics	What project metrics will be collected and archived?	
Project Artifacts	What project artifacts will be collected and examined during the retrospective?	
Communicating Plans	How and by whom will the planning details be communicated to the participants?	
Individual Preparation	What individual preparation is necessary prior to the retrospective meeting?	
Logistics	What items does the facilitator need to take to the retrospective meeting (pens, markers, sticky notes, tape, transparencies, paper, laptop)? Is the room properly equipped (projectors, flipcharts, space)?	
Communicating Results	How will the meeting results be communicated? Who will receive the communication? Consider objectives, scope, deliverables.	
Action Plan Owner	Who will be responsible for follow-up on action plans and process improvements following the retrospective?	

References

Armour, Frank, and Granville Miller. 2001. *Advanced Use Case Modeling: Software Systems*. Boston, Mass.: Addison-Wesley.

Boddie, John. 1995. "Growing Better Managers by Doing Nothing." *American Programmer*, 8(1): 21-27.

Boehm, Barry, and Ronnie Ross. 1989. "Theory-W Software Project Management: Principles and Examples." *IEEE Transactions on Software Engineering,* 15(7): 902-916.

Brackett, John W. 2001. "Planning and Executing Second-Generation E-Projects." Cutter Consortium *E-Project Management Advisory Service Executive Report,* 2(6).

Brooks, Jr., Frederick P. 1995. *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition.* Boston, Mass.: Addison-Wesley.

Brown, Norm. 1996. "Industrial-Strength Management Strategies." *IEEE Software*, 13(4): 94-103.

Brown, Norm. 1999. "High-Leverage Best Practices: What Hot Companies Are Doing to Stay Ahead." *Cutter IT Journal*, 12(9): 4-9.

Brown, William J., Raphael C. Malveau, Hays W. McCormick III, and Thomas J. Mowbray. 1998. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. New York: John Wiley & Sons.

Chrissis, Mary Beth, Mike Konrad, and Sandy Shrum. 2003. *CMMI: Guidelines for Process Integration and Product Improvement*. Boston, Mass.: Addison-Wesley.

Collier, Bonnie, Tom DeMarco, and Peter Fearey. 1996. "A Defined Process for Project Postmortem Review." *IEEE Software,* 13(4): 65-72.

Covey, Stephen R. 1989. *The 7 Habits of Highly Effective People.* New York: Fireside.

DeMarco, Tom. *Structured Analysis and System Specification*. 1979. Englewood Cliffs, N.J.: Prentice-Hall.

Fisher, Roger, William Ury, and Bruce Patton. 1991. *Getting to Yes: Negotiating Agreement Without Giving In, 2nd Edition*. New York: Penguin Books.

Fogle, Sam, Carol Loulis, and Bill Neuendorf. 2001. "The Benchmarking Process: One Team's Experience." *IEEE Software*, 18(5): 40-47.

Foody, Michael A. 1995. "When Is Software Ready For Release?" *Unix Review,* 13(3):35-41.

Gilb, Tom. 2005. *Competitive Engineering: A Handbook for Systems Engineering, Requirements Engineering, and Software Engineering Using Planguage.* Burlington, Mass.: Butterworth-Heinemann.

Gottesdiener, Ellen. 2001. "Decide How to Decide." *Software Development*, 9(1): 65-70.

Grady, Robert B., and Deborah L. Caswell. 1987. *Software Metrics: Establishing a Company-Wide Program.* Englewood Cliffs, N.J.: Prentice-Hall.

Highsmith, Jim. 2004. *Agile Project Management: Creating Innovative Products.* Boston, Mass.: Addison-Wesley.

Humphrey, Watts. 1997. *Managing Technical People: Innovation, Teamwork, and the Software Process.* Reading, Mass.: Addison-Wesley.

International Function Point Users Group (IFPUG). 2002. *Function Point Counting Practices Manual, Version 4.1.1*. Princeton Junction, N.J.: International Function Point Users Group.

Jeffries, Ron, Ann Anderson, and Chet Hendrickson. 2001. *Extreme Programming Installed*. Boston, Mass.: Addison-Wesley.

Karten, Naomi. 1994. *Managing Expectations: Working with People Who Want More, Better, Faster, Sooner, NOW!* New York: Dorset House Publishing.

Kerth, Norman L. 2001. *Project Retrospectives: A Handbook for Team Reviews*. New York: Dorset House Publishing.

Kulak, Daryl, and Eamonn Guiney. 2004. *Use Cases: Requirements in Context, 2nd Edition*. Boston, Mass.: Addison-Wesley.

May, Lorin J. 1998. "Major Causes of Software Project Failures." *CrossTalk,* 11(7): 9-12.

McConnell, Steve. 1996. *Rapid Development: Taming Wild Software Schedules*. Redmond, Wash.: Microsoft Press.

McManus, John. 2005. *Managing Stakeholders in Software Development Projects.* Oxford, England: Elsevier Butterworth-Heinemann.

Moore, Geoffrey A. 1991. *Crossing the Chasm: Marketing and Selling High-Tech Products to Mainstream Customers*. New York: HarperBusiness.

Musa, John D., Anthony Iannino, and Kazuhira Okumoto. 1987. *Software Reliability: Measurement, Prediction, Application*. New York: McGraw-Hill.

Nejmeh, Brian A., and Ian Thomas. 2002. "Business-Driven Product Planning Using Feature Vectors and Increments." *IEEE Software,* 19(6): 34-42.

Paulk, Mark, et al. 1995. *The Capability Maturity Model: Guidelines for Improving the Software Process.* Reading, Mass.: Addison-Wesley.

Project Management Institute. 2000. *A Guide to the Project Management Body of Knowledge, 2000 Edition*. Newtown Square, Pa.: Project Management Institute.

Rogers, Everett M., and Shoemaker, F. Floyd. 1971. *Communication of Innovation: A Cross-Cultural Approach*. New York: The Free Press.

Rothman, Johanna. 2002. "Is This Software Done?" *STQE*, 4(2): 30-35.

Sassenburg, Hans. 2003. "When Can the Software Be Released?" *Proceedings of the European SEPG*, London.

Simmons, Erik. 2001. "From Requirements to Release Criteria: Specifying, Demonstrating, and Monitoring Product Quality." *Proceedings of the Pacific Northwest Software Quality Conference*. Portland, Ore..

Smith, Larry W. 2000. "Project Clarity Through Stakeholder Analysis." *CrossTalk*, 13(12): 4-9.

Verner, June M., and William M. Evanco. 2005. "What Project Management Practices Lead to Success?" *IEEE Software,* 22(1): 86-93.

Voas, Jeffrey. 2001. "Faster, Better, and Cheaper." *IEEE Software,* 18(3): 96-97.

Walsh, James. 1993. "Determining Software Quality." *Computer Language,* 10(4): 57-65.

Weinberg, Gerald M. 1985. *The Secrets of Consulting: A Guide to Giving & Getting Advice Successfully*. New York: Dorset House Publishing.

Whitmire, Scott A. 1995. "An Introduction to 3D Function Points." *Software Development,* 3(4): 43–53.

Whitten, Neal. 1995. *Managing Software Development Projects: Formula for Success, 2nd Edition*. New York: John Wiley & Sons.

Wiegers, Karl E. 1996. *Creating a Software Engineering Culture.* New York: Dorset House Publishing.

Wiegers, Karl E. 2002a. *Peer Reviews in Software: A Practical Guide*. Boston, Mass.: Addison-Wesley.

Wiegers, Karl. 2002b. "Promises, Promises." *TheRationalEdge.com* (January).

Wiegers, Karl. 2002c. "Saving for a Rainy Day." *TheRationalEdge.com* (April).

Wiegers, Karl. 2002d. "Success Criteria Breed Success." *TheRationalEdge.com* (February).

Wiegers, Karl E. 2003. *Software Requirements, 2nd Edition*. Redmond, Wash.: Microsoft Press.

Wiegers, Karl, and Johanna Rothman. 2001 "Looking Back, Looking Ahead." *Software Development*, 9(2): 65-69.

Wilson, Peter B. 1995. "Testable Requirements—An Alternative Sizing Measure." *The Journal of the Quality Assurance Institute,* 9(4): 3–11.

Wysocki, Robert K., and Rudd McGary. 2003. *Effective Project Management: Traditional, Adaptive, Extreme, 3rd Edition.* New York: Wiley Publishing, Inc.

Acknowledgments

I appreciate the many helpful review comments provided by Chris Fahlbusch, Ellen Gottesdiener, Andre Gous, Ulla Merz, Johanna Rothman, and Scott Whitmire.

About Process Impact

Process Impact is a software process consulting and education company in Portland, Oregon. Karl Wiegers, Ph.D., the Principal Consultant, has many years of experience with software development and management. His interests include project management, requirements engineering, peer reviews, process improvement, and metrics. Karl is the author of the books *Software Requirements, 2nd Edition*, *More About Software Requirements*, *Peer Reviews in Software,* and *Creating a Software Engineering Culture*. Process Impact provides a variety of training and consulting services. Visit http://www.processimpact.com for many magazine articles, templates, spreadsheet tools, other process assets, and tasty recipes (on the Biography page).

Training Seminars

Process Impact's presentations include the following full seminars, in addition to many shorter presentations. Seminars can be customized to meet your needs. You may license these seminars for presentation by your own staff.

"Project Management Best Practices" (1 day)
"Introduction to Software Risk Management" (half-day)
"In Search of Excellent Requirements" (1-3 days)
"Exploring User Requirements with Use Cases" (1 day)
"Writing Quality Requirements Workshop" (1-day)
"Software Inspections and Peer Reviews" (1 day)
"Creating a Software Engineering Culture" (1 day)
"Software Subcontract Management Process and Guidance" (1.5 days)

eLearning Seminars

Several of our most popular seminars are available on CD in a rich, self-paced eLearning format for both individual and site licensing. These courses integrate animated slides with narration by Karl Wiegers in a presentation that plays in your Web browser. They include slide notes, student handouts, quizzes, practice sessions, magazine articles, templates, and other process assets.

"In Search of Excellent Requirements" (10 hours of audio)
"Exploring User Requirements with Use Cases" (6 hours of audio)
"Project Management Best Practices" (7.5 hours of audio)
"Software Inspections and Peer Reviews" (6 hours of audio)
"Process Impact Virtual Conference" (6.5 hours of audio)
"Software Requirements: An Executive Overview" (90 minutes of audio)

Consulting

Process Impact provides both on-site and off-site consulting services to software organizations of all types. Let us help you tune up your software processes, write better requirements specifications, and manage projects more successfully.

Contact Information

Process Impact
11491 SE 119th Drive
Clackamas, OR 97015-8778
503-698-9620 (voice) 503-698-9680 (fax)
sales@processimpact.com
http://www.processimpact.com