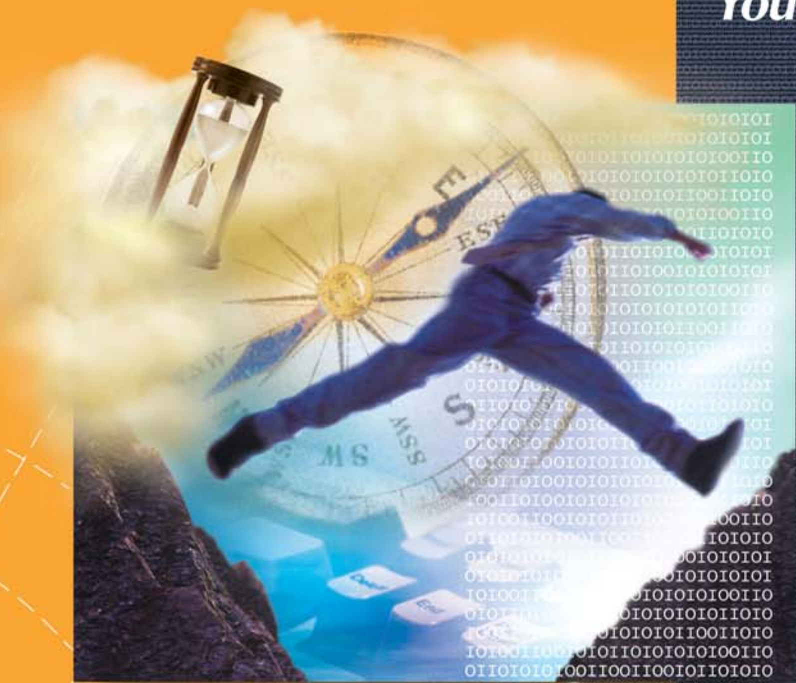
SOFTWARE PROJECT SURVIVAL GUIDE

Microsoft

How to Be Sure Your First Important Project Isn't Your Last



Steve McConnell

Author of *Code Complete* and *Rapid Development*

Best Practices

Here is Edward Bear, coming downstairs now, bump, bump, bump on the back of his head, behind Christopher Robin.
It is, as far as he knows, the only way of coming downstairs,
but sometimes he feels that there really is another way,
if only he could stop bumping for a moment and think of it.
And then he feels that perhaps there isn't.

Software Project Survival Guide

BY

Steve McConnell

Microsoft Press

Software Project Survival Guide

Published by Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Cataloging-in-Publication Data
McConnell, Steve.
Software Project Survival Guide : how to be sure your first important project isn't your last / Steve McConnell.
p. cm.
Includes index.
ISBN 978-0-735-69148-3
1. Computer software--Development--Management. I. Title.
QA76.76.D47M394 1997
005.1'068'4--dc21 97-37923
CIP

Printed and bound in the United States of America.

16 17 18 19 20 QWE 7 6 5 4 3 2

Distributed in Canada by H.B. Fenn and Company Ltd.

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at www.microsoft.com/mspress. Send comments to *mspinput@microsoft.com*.

Acquisitions Editor: David Clark
Project Editor: Victoria Thulman

Part No. 097-0008675

Contents

Acknowledgments

As an experiment, I posted draft chapters of this book on my Internet Web site and invited readers to comment on them. Many people downloaded the chapters, and they contributed literally thousands of insightful review comments. The diversity of viewpoints was tremendous (bordering on overwhelming), and the book is more readable, cohesive, practical, and useful as a result.

Thanks first to the people who reviewed the whole manuscript. These people include Robert C. Burns (The Boeing Company), Lawrence Casey, Alan Brice Corwin (Process Builder), Thomas Duff, Mike Cargal, Pat Forman (Lynden), Manny Gatlin, Marc Gunter, Tom Hill, William Horn, Greg Hitchcock, Grant McLaughlin, Mike Morton, Matt Peloquin, David Roe, Steve Rinn, André Sintzoff, Matthew J. Slattery, and Beth Weiss.

I am also grateful to the people who commented on significant sections of the book, including Ray Bernard (Ray Bernard Consulting and Design), Steven Black, Robert Brown, Jacob L. Cybulski, Tom Gilb, Dick Holland, Gerard Kilgallon, Art Kilner, Steve Kobb, Robert E. Lee, Pete Magsig, Hank Meuret (Meuret Consulting), Al Noel, Karsten M. Self, Rob Thomsett, and Gregory V. Wilson.

Other people commented on one or more details of the manuscript, and I've listed those people where appropriate in the "Notes" section at the end of the book.

It was a pleasure to see the staff at Microsoft Press transform the raw material of my manuscript into finished form. Special thanks to Victoria Thulman, project editor, for her wonderful forbearance and resiliency in accommodating an author who has opinions about every facet of book production. Thanks to Kim Eggleston for the book's spare, elegant design, and to the rest of the Microsoft Press staff, including David Clark, Abby Hall, Cheryl Penner, and Michael Victor.

Thanks finally to my wife, Tammy, for her unmatchable moral support and trademark good humor. (This is number three, so now you have to think of a new joke. Fa!)

Preliminary Survival Briefing

About two million people are working on about 300,000 software projects in the United States at this time.[1] Between one third and two thirds of those projects will exceed their schedule and budget targets before they are delivered. Of the most expensive software projects, about half will eventually be canceled for being out of control. Many more are canceled in subtle ways—they are left to wither on the vine, or their sponsors simply declare victory and leave the battlefield without any new software to show for their trouble. Whether you're a senior manager, an executive, a software client, a user representative, or a project leader, this book explains how to prevent your project from suffering these consequences.

Software projects fail for one of two general reasons: the project team lacks the knowledge to conduct a software project successfully, or the project team lacks the resolve to conduct a project effectively. This book cannot do much about the lack of resolve, but it does contain much of the knowledge needed to conduct a software project successfully.

The factors that make a software project successful are not especially technical. Software projects are sometimes viewed as mysterious entities that survive or perish based on the developers' success in chanting magic technical incantations. When asked why they delivered a component two weeks late, developers say things like, "We had to implement a 32-bit thunking layer to interface with our OCX interface." Faced with explanations like that, it is no wonder that people without deep technical expertise feel powerless to influence a software project's success.

1. Source citations and notes about related topics can be found in the "Notes" section at the end of the book.

The message of the *Software Project Survival Guide* is that software projects survive not because of detailed technical considerations like "thunking layers" but for much more mundane reasons. Software projects succeed or fail based on how carefully they are planned and how deliberately they are executed. The vast majority of software projects can be run in a deterministic way that virtually assures success. If a project's stakeholders understand the major issues that determine project success, they can ensure that their project reaches a successful conclusion. The person who keeps a software project headed in the right direction can be a technical manager or an individual software developer—it can also be a top manager, a client, an investor, an end-user representative, or any other concerned party.

Who Should Read This Book

This book is for anyone who has a stake in a software project's outcome.

Top Managers, Executives, Clients, Investors, and End-User Representatives

Nonsoftware people are commonly given responsibility for overseeing the development of a software product. These people have backgrounds in sales, accounting, finance, law, engineering, or some other field. They might not have any formal authority to direct the project, but they will still be held accountable for seeing that the project goes smoothly. At a minimum, they are expected to sound an alarm if the project begins to go awry.

If you're in this group, this book will provide you with a short, easily readable description of what a successful project looks like. It will give you many ways to tell in advance whether the project is headed for failure or success. It will also describe how to tell when no news is good news, when good news is bad news, or when good news really is good news.

Project Managers

Many software project managers are thrust into management positions without any training specific to managing software projects. If you're in this group, this book will help you master the key technical management skills of requirements management, software project planning, project tracking, quality assurance, and change control.

Technical Leaders, Professional Developers, and Self-Taught Programmers

If you're an expert in technical details, you might not have had much exposure to the big-picture issues that project leaders need to focus on. In that case, you can think of this book as an annotated project plan. By providing an overview of a successful software project, this book will help you make the transition from expert technician to effective project leader. You can use the plan described in this book as a starting point, and you can tailor its strategies to the needs of your specific projects. If you've read *Rapid Development*, the first part of this book will be about half review for you. You might want to skim Chapters 1 through 5, read the end of Chapter 5 carefully, skim Chapter 6, and then begin reading carefully again starting with Chapter 7.

Kinds of Projects This Book Covers

The plan will work for business systems, broad-distribution shrink-wrap software, vertical market systems, scientific systems, and similar programs. It is designed for use on desktop client/server projects using modern development practices such as object-oriented design and programming. It can easily be adapted for projects using traditional development practices and mainframe computers. The plan has been designed with project team sizes of 3 to 25 team members and schedules of 3 to 18 months in mind. These are considered to be medium-sized projects. If your project is smaller you can scale back some of this book's recommended practices. (Throughout the book, I point out places you can do that.)

This book is primarily intended for projects that are currently in the planning stages. If you're at the beginning of the project, you can use the approach as the basis for your project plan. If you're in the middle of a project, the Survival Test in Chapter 2 and the Survival Checks at the end of each chapter will help you determine your project's chance of success.

By itself, this book's plan is not formal or rigorous enough to support life-critical or safety-critical systems. It is appropriate for commercial applications and business software, and it is a marked improvement over many of the plans currently in use on multimillion-dollar projects.

A Note to Advanced Technical Readers

The *Software Project Survival Guide* describes one effective way to conduct a software project. It is not the only effective way to run a project, and for any specific project it might not be the optimum way. The extremely knowledgeable technical leader will usually be able to come up with a better, fuller, more customized development plan than the generic one described here. But the plan described here will work much better than a hastily thrown together plan or no plan at all, and no plan at all is the most common alternative.

The plan described in the following chapters has been crafted to address the most common weaknesses that software projects face. It is loosely based on the "key process areas" identified by the Software Engineering Institute (SEI) in Level 2 of the SEI Capability Maturity Model. The SEI has identified these key processes as the critical factors that enable organizations to meet their schedule, budget, quality, and other targets. About 85 percent of all organizations perform below Level 2, and this plan will support dramatic improvements in those organizations. The SEI has defined the key process areas of Level 2 as follows:

- Project planning
- Requirements management
- Project tracking and oversight
- Configuration management
- Quality assurance
- Subcontract management

This book addresses all of these areas except subcontract management.

This Book's Foundation

In writing this book, I have kept three primary references at my elbow that have been invaluable resources, in addition to the many other resources I've drawn from. I've tried to condense the contents of these three references and present them in the most useful way that I can.

The first reference is the Software Engineering Institute's *Key Practices of the Capability Maturity Model, Version 1.1*. This book is a gold mine of hard-won industry experience in prioritizing implementation of new development practices. At almost 500 pages it is somewhat long, and even at that length the information is still dense. It is not a tutorial and so is not intended for the novice reader. But for someone who has a basic understanding of the practices it describes, the summary and structure that *Key Practices* provides is a godsend. This book is available free on the Internet at *http://www.sei.cmu.edu/* or from the National Technical Information Service (NTIS) branch of the U.S. Department of Commerce in Springfield, Virginia.

The second reference is the NASA Software Engineering Laboratory's (SEL's) *Recommended Approach to Software Development, Revision 3*. The SEL was the first organization to receive the IEEE Computer Society's Process Achievement Award. Many keys to the success of its process are described in the *Recommended Approach*. Whereas the SEI's document describes a set of practices without showing how to apply them to a specific project, the *Recommended Approach* describes a structured sequence of practices. The two volumes together form a complementary set. This book is also available free on the Internet at *http://fdd.gsfc.nasa.gov/seltext.html.*

The final "book" at my elbow has been my own experience. I am writing not as an academician who wants to create a perfect theoretical framework, but as a practitioner who wants to create a practical reference that will aid me in my work and my clients in theirs. The information drawn together here will make it easier for me to plan and conduct my next project and easier to explain its critical success factors to my clients. I hope it does the same for you.

Steve McConnell
Bellevue, Washington
August 1997

3 Survival Concepts

Well-defined development processes are important and necessary elements of software project survival. With well-defined processes, software personnel can spend most of their time on productive work that moves the project steadily toward completion. With poorly planned processes, developers spend a lot of their time correcting mistakes. Much of the leverage for project success is contained in upstream activities, and knowledgeable software stakeholders ensure that projects focus enough attention on upstream activities to minimize problems downstream.

Before you begin a mission, you are briefed about its most important characteristics. This chapter describes the critical factors that contribute to software mission success.

The Power of "Process"

This book is about using effective software development processes. The phrase "software processes" can mean a lot of different things. Here are some examples of what I mean by "software processes:"

- Committing all requirements to writing.
- Using a systematic procedure to control additions and changes to the software's requirements.
- Conducting systematic technical reviews of all requirements, designs, and source code.
- Developing a systematic Quality Assurance Plan in the very early stages of the project that includes a test plan, review plan, and defect tracking plan.
- Creating an implementation plan that defines the order in which the software's functional components will be developed and integrated.
- Using automated source code control.
- Revising cost and schedule estimates as each major milestone is achieved. Milestones include the completion of requirements analysis, architecture, and detailed design as well as the completion of each implementation stage.

These processes are beneficial in ways that will shortly become apparent.

Negative View of Process

The word "process" is viewed as a four-letter word by some people in the software development community. These people see "software processes" as rigid, restrictive, and inefficient. They think that the best way to run a project is to hire the best people you can, give them all the resources they ask for, and turn them loose to let them do what they're best at. According to this view, projects that run without any attention to process can run extremely efficiently. People who hold this view imagine that the relationship between work and productivity over the course of a project looks like the chart shown in Figure 3-1 on the facing page.

People who hold this view acknowledge that some amount of "thrashing," or unproductive work, will take place. Developers will make mistakes, they agree, but they will be able to correct them quickly and efficiently—certainly at less overall cost than the cost of "process."

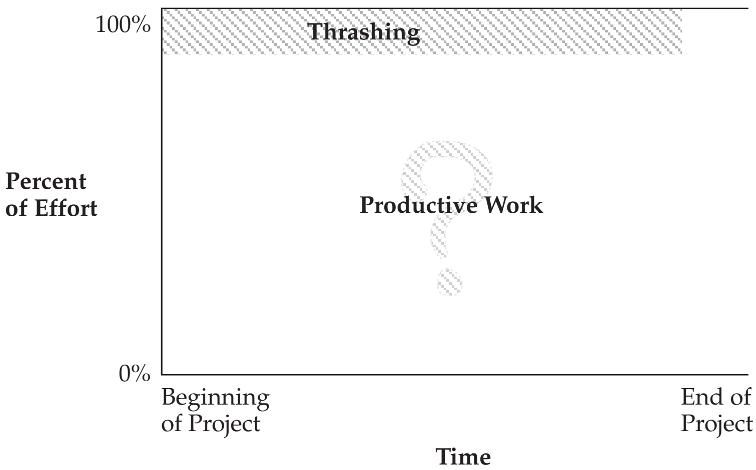


FIGURE 3-1 *Mistaken perception that ignoring process increases the proportion of productive work on projects.*

Adding process, then, is thought to be pure overhead and simply takes time away from productive work, as shown in Figure 3-2.

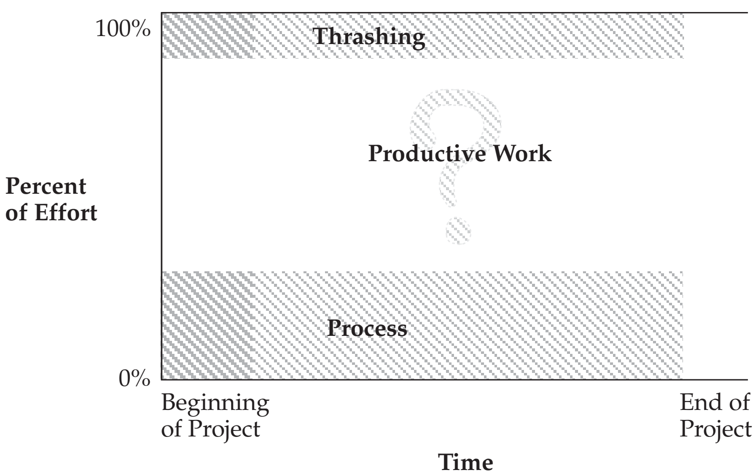


FIGURE 3-2 *Mistaken perception that an attention to process will decrease the proportion of productive work. (Process is seen as pure overhead.)*

This point of view has intuitive appeal. At the beginning of a project (shown by the darker shaded areas), a focus on process certainly does take time away from productive work. If that trend continued throughout a project (shown by the lighter shaded areas), it wouldn't make sense to spend much time on process.

Software industry experience with medium-size projects, however, has revealed that the trend shown in Figure 3-2 does not continue throughout the project. Projects that don't pay attention to establishing effective processes early on are forced to slap them together later, when slapping them together takes more time and does less good. Here are some scenarios that illustrate why earlier is better:

- *Change control.* In the middle of the project, team members informally agree to implement a wide variety of changes that are directly proposed to them by their manager or customer. They don't begin controlling changes systematically until late in the project. By that time, the scope of the product has expanded by 25 to 50 percent or more, and the budget and the schedule have expanded accordingly.
- *Quality assurance.* Projects that don't set up processes to eliminate defects in early stages fall into extended test-debug-reimplement-retest cycles that seem interminable. So many defects are reported by testing that by the end of the project, the "change control board" or "feature team" may be meeting as often as every day to prioritize defect corrections. Because of the vast number of defects, the software has to be released with many known (albeit low priority) defects. In the worst case, the software might never reach a level of quality high enough for it to be released.
- *Uncontrolled revisions.* Major defects discovered late in the project can cause the software to be redesigned and rewritten during testing. Since no one planned to rewrite the software during testing, the project deviates so far from its plans that it essentially runs without any planning or control.

- *Defect tracking.* Defect tracking isn't set up until late in the project. Some reported defects are not fixed simply because they are forgotten, and the software is released with these defects even though they would have been easy to fix.
- *System integration.* Components developed by different developers are not integrated with one another until the end of the project. By the time the components are integrated, the interfaces between components are out of synch and much work must be done to bring them back into alignment.
- *Automated source code control.* Source code revision control isn't established until late in the project, after developers have begun to lose work by accidentally overwriting the master copies of their own or one another's source code files.
- *Scheduling.* On projects that are behind schedule, developers are asked to reestimate their remaining work as often as once a week or more, taking time away from their development work.

When a project has paid too little early attention to the processes it will use, by the end of a project developers feel that they are spending all of their time sitting in meetings and correcting defects and little or no time extending the software. They know the project is thrashing. When developers see they are not meeting their deadlines, their survival impulses kick in and they retreat to "solo development mode," focusing exclusively on their personal deadlines. They withdraw from interactions with managers, customers, testers, technical writers, and the rest of the development team, and project coordination unravels.

Far from a steady level of productive work suggested by Figure 3-1, the medium-size project conducted without much attention to development processes typically experiences the pattern shown in Figure 3-3 on the next page.

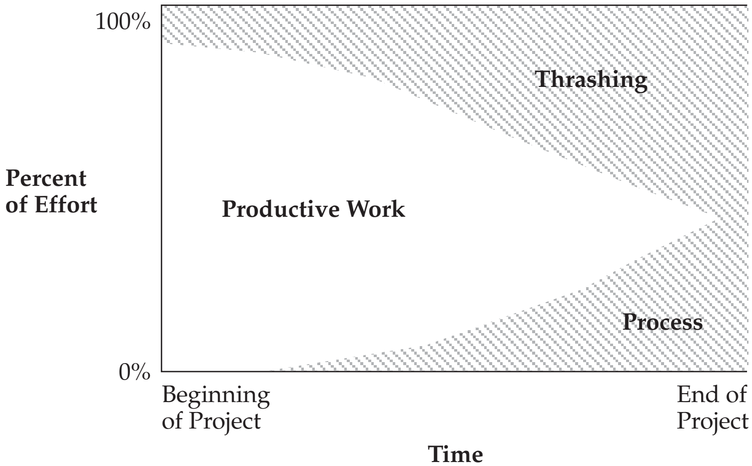


FIGURE 3-3 *Real experience of projects that pay little attention to process. As the project environment becomes increasingly complicated, thrashing and process both increase.*

In this pattern, projects experience a steady increase in thrashing over the course of a project. By the middle of the project, the team realizes that it is spending a lot of time thrashing and that some attention to process would be beneficial. But by then much of the damage has been done. The project team tries to increase the effectiveness of its process, but its efforts hold the level of thrashing steady, at best. In some cases, the late attempt to improve the project's processes actually makes the thrashing worse.

The lucky projects release their software while they are still eking out a small amount of productive work. The unlucky projects can't complete their software before reaching a point at which 100 percent of their time is spent on process and thrashing. After spending several weeks or months in this condition, such a project is typically canceled when management or the customer realizes that the project is no longer moving forward. If you think that attention to process is needless overhead, consider that the overhead of a canceled project is 100 percent.

PROCESS TO THE RESCUE

Fortunately, there are a variety of alternatives to this dismal scenario, and the best do not rely at all on rigid, inefficient processes (also known as R.I.P.). Some processes certainly are rigid and inefficient, but I don't recommend that projects use them. The approach described in this book requires use of processes that *increase* the project's flexibility and efficiency.

When these kinds of processes are used, the project profile looks like the one shown in Figure 3-4.

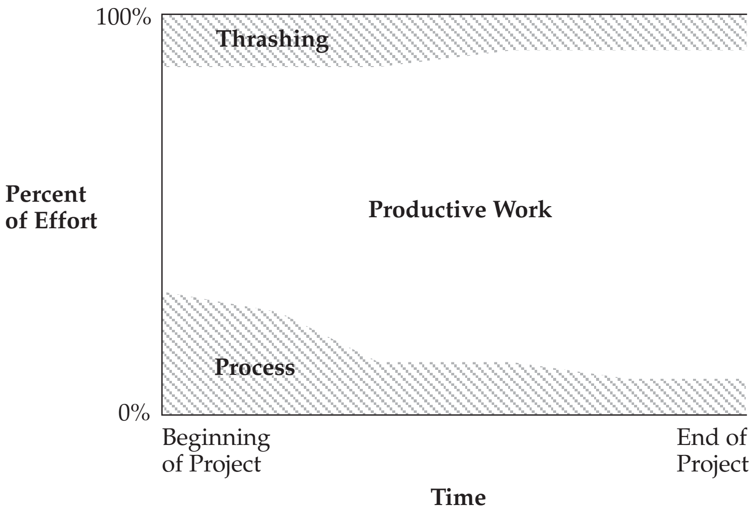


FIGURE 3-4 *Experience of projects that focus early attention on process. As the team gains experience with its processes and fine tunes them to the working environment, the time spent on process and thrashing both diminish.*

During the first few weeks of the project, the process-oriented team will seem less productive than the process-phobic team because the level of thrashing will be the same on both projects, and the process-oriented team will be spending a significant amount of its time on processes.

By the middle of the project, the team that focused on process early will have reduced the level of thrashing compared to the beginning of the project, and will have streamlined its processes. At that point, the process-phobic team will be just beginning to realize that thrashing is a significant problem and just beginning to institute some processes of its own.

By the end of the project, the process-oriented team will be operating at a high-speed hum, with little thrashing, and it will be performing its processes with little conscious effort. This team tolerates a small amount of thrashing because eliminating the last bit of thrashing would cost more in overhead than would be saved. When all is said and done, the overall effort on the project will be considerably lower than the effort of the process-phobic team.

An investment made in process at the beginning of the project produces large returns later in the project.

Organizations that have explicitly focused on improving their development processes have, over several years, cut their time-to-market by about one-half and reduced their costs and defects by factors of 3 to 10. Over a 5-year period, Lockheed cut its development costs by 75 percent, reduced its time-to-market by 40 percent, and reduced its defects by 90 percent. Over a 6.5-year period, Raytheon tripled its productivity and realized a return on investment (ROI) in process improvement of almost 8 to 1. Bull HN realized an ROI of 4 to 1 after 4 years of software process improvement efforts, and Schlumberger realized an ROI of almost 9 to 1 after 3.5 years of software process improvement. NASA's Software Engineering Laboratory cut its average cost per mission by 50 percent and its defect rate by 75 percent over an 8-year period while dramatically increasing the complexity of software used on each mission. Similar results have been reported at Hughes, Loral, Motorola, Xerox and other companies that have focused on systematically improving their software processes.

Here's the best news: Can you guess the average cost of these improvements in productivity, quality, and schedule performance? It's about 2 percent of total development costs—typically about $1,500 per developer per year.

Process vs. Creativity and Morale

One of the common objections to putting systematic processes in place is that they will limit programmers' creativity. Programmers do indeed have a high need to be creative. Managers and project sponsors also have a need for projects to be predictable, to provide progress visibility, and to meet schedule, budget, and other targets.

The criticism that systematic processes limit developers' creativity is based on the mistaken idea that there is some inherent contradiction between developers' creativity and the satisfaction of management objectives. It is certainly possible to create an oppressive environment in which programmer creativity and management goals are placed at odds, and many companies have done that, but it is just as possible to set up an environment in which those goals are in harmony and can be achieved simultaneously.

Companies that have focused on process have found that effective processes support creativity and morale. In a survey of about 50 companies, only 20 percent of the people in the least process-oriented companies rated their staff morale as "good" or "excellent." In organizations that paid more attention to their software processes, about 50 percent of the people rated their staff morale as good or excellent. And in the most process-sophisticated organizations, 60 percent of the people rated their morale as good or excellent.

Programmers feel best when they're most productive. Good project leadership establishes a clear vision and then puts a process framework into place that allows programmers to feel incredibly productive. Programmers dislike weak leadership that provides too little structure because they end up working at cross purposes and, inevitably, are forced to throw away huge chunks of their work. Programmers appreciate enlightened leadership that emphasizes predictability, visibility, and control.

The appropriate response to the so-called contradiction between process and creativity is that none of the processes described in this book will limit programmers' creativity in any way that matters. Most provide a supporting structure that will free programmers to be more creative about the technical work that matters and free them from the distractions that typically consume their attention on poorly run projects.

Transitioning to a Systematic Process

If a project team isn't currently using a systematic process, one of the easiest ways to transition to one is to map out the current software development process, identify the parts of that process that aren't working, and then try to fix those parts. Although project teams will sometimes claim that they don't currently have a process, every project team has a process of some kind. (If they claim not to have one, they probably just don't have a very good one.)

The least sophisticated process typically looks like this:

1. Discuss the software that needs to be written.
2. Write some code.
3. Test the code to identify the defects.
4. Debug to find root causes of defects.
5. Fix the defects.
6. If the project isn't done yet, return to step 1.

This book describes a more sophisticated and more effective software process.

One obstacle to creating a systematic software process is that project teams are afraid they will err on the side of having too much process—that their process will be overly bureaucratic and create too much overhead for the project. This is typically not a significant risk for several reasons:

- A project that uses the approach described in this book will have a fairly sophisticated process without incurring much overhead.
- Software projects are often larger than they at first appear. Far more projects err on the side of too little process than too much.
- Starting with too much process and loosening some of the processes later on, if needed, is easier than starting with too little process and trying to add additional processes once a project is under way.
- The cost and schedule penalty for having too much process is far smaller than the penalty for having too little process, for reasons I will explain next.

Upstream, Downstream

Good software processes are designed to root out problems early in the project. This concept is important enough to discuss in some detail.

You'll sometimes hear experienced software developers talk about the "upstream" and "downstream" parts of a software project. The word "upstream" simply refers to the early parts of a project such as requirements development and architecture, and "downstream" refers to the later parts such as construction and system testing.

I have found that this distinction between “upstream” and “downstream” is a fundamentally useful way to think about a software project. The work developers do early in the project is placed into a stream and has to be fished back out later in the project. If the early work is done well, the work that’s fished out later is healthy and contributes to project success. If the early work is done poorly, the work that’s fished out later can severely impair the project. In extreme circumstances, it can prevent the project from ever getting finished.

Researchers have found that an error inserted into the project stream early—for example, an error in requirements specification or architecture—tends to cost 50 to 200 times as much to correct late in the project as it does to correct close to the point where it was originally put into the stream. Figure 3-5 illustrates this effect.

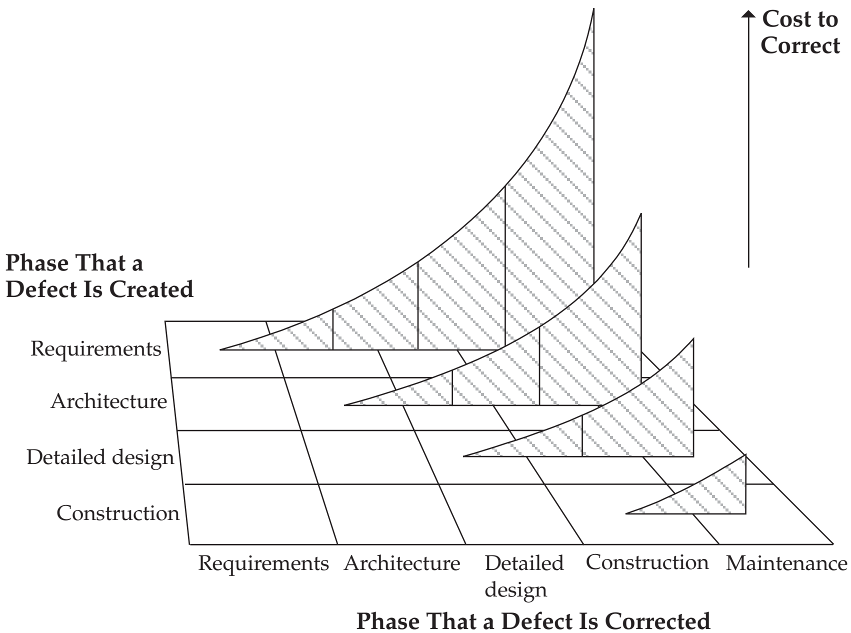


FIGURE 3-5 *Increase in defect cost as time between defect creation and defect correction increases. Effective projects practice “phase containment”—the detection and correction of defects in the same phase in which they are created.*

One sentence in a requirements specification can easily turn into several design diagrams. Later in the project, those diagrams can turn into hundreds of lines of source code, dozens of test cases, many pages of end-user documentation, help screens, instructions for technical support personnel, and so on.

If the project team has an opportunity to correct a mistake at requirements time when the only work that has been done is the creation of a one-sentence requirements statement, it makes good sense for the team to correct that statement rather than to correct all the various manifestations of the inadequate requirements statement downstream. This idea is sometimes called "phase containment," and refers to the detection and correction of defects in the same phase in which the defects are introduced.

Successful project teams create their own opportunities to correct upstream problems by conducting thorough, careful reviews of requirements and architecture.

Because no code is generated while the upstream activities are conducted, these activities might seem as though they are delaying "the real work" of the project. In reality, they are doing just the opposite. They are laying the groundwork for the project's success.

Erring on the side of too much process will marginally increase the project's overhead, but erring on the side of too little allows defects to slip through that must be corrected at 50 to 200 times the efficient cost of correcting them. For this reason, the smart money errs on the side of too much process rather than on the side of too little.

Cone of Uncertainty

One of the reasons that mistakes made early in a project cost 50 to 200 times as much to correct downstream as upstream is that the upstream decisions tend to be farther reaching than the downstream decisions.

Early in the project, a project team addresses the large issues like whether to support Windows NT and the Macintosh or just Windows NT, and whether to provide fully customizable reports or fixed format reports. In the middle of the project, a project team addresses medium-size issues, such as how many subsystems to have, how in general to handle error-processing, and how to adapt a printing routine from an old project to the current project. Late in the project, a project team addresses small issues, such as which technical algorithm to use and whether to allow the user to cancel an operation when it's partway complete. As the cone of uncertainty in Figure 3-6 suggests, software development is a process of continuous refinement, which proceeds from large grain to small grain, from large decisions to small decisions. The time burned on a software project is the time required to think through and make these decisions. Decisions made at one stage of the project affect the next set of decisions.

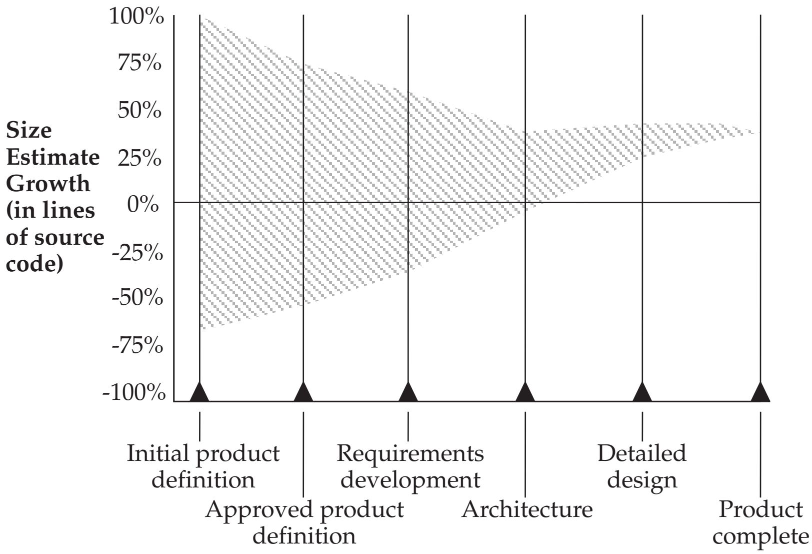


FIGURE 3-6 *Cone of uncertainty. Decision-making on a software project progresses from large grain to small grain. The project team can't know much about the decisions to be made in a specific phase until it has completed most of the work for the phase that immediately precedes it.*

Before the project team has actually made the first set of decisions, it can only make the most general educated guess about the decisions that will be made later in the project. After the set of decisions at one level of granularity have been made, a team can make pretty accurate estimates of the kinds of decisions that will need to be made at the next level of granularity. The project team makes the best decisions it can at the large-grain level, but sometimes unforeseen (and unforeseeable) issues at the fine-grain level percolate back up to a larger context, and the need to cancel an operation when it's partway complete means that the project team has to redesign a routine, a module, or a subsystem.

If you want to understand what software development is all about, you need to understand that the project team has to think through and make all the decisions in one stage before it can know enough about the next stage even to estimate the work involved in it.

Implications for Project Estimation

The cone of uncertainty has strong implications for software project estimation. It implies that it is not only difficult to estimate a project accurately in the early stages, it is *theoretically impossible*. At the end of the requirements development phase, the scope of the project will be determined by myriad decisions yet to be made during architecture, detailed design, and construction. The person who claims to be able to *estimate* the impact of those myriad decisions before they are actually made is either a prophet or not very well informed about the intrinsic nature of software development.

On the other hand, the person who seeks to *control* the way those decisions are made in order to meet the project's schedule or budget targets is operating sensibly. You can set firm schedule and budget targets early in the project as long as you're willing to be ruthless about cutting planned functionality to meet those targets. Keys to success in meeting targets in this way include setting crystal clear and non-conflicting goals at the beginning of the project, keeping the product concept very flexible, and then actively tracking and controlling development work throughout the rest of the project.

Early in the project you can have
firm cost and schedule targets
or a firm feature set, but not both.

Survival Check

- Project leadership understands the critical role of well-defined processes and supports them.
- The project's processes are generally oriented toward detecting as many problems upstream as possible.
- Project leadership recognizes that estimates made during the first half of the project are inherently imprecise and will need to be refined as the project progresses.

INDEX

Page numbers in italics refer to tables, figures, or illustrations.

D

E

F

G

H

I

L

M

N

O

P

Q

R

S