“Most progressive organizations strive to increase their overall performance by inserting either advanced technologies or tight processes. What tends to be missed is the enlightened leader that can motivate and build an environment that leverages the technology and processes introduced. Some processes and technologies allow excellent teams to excel, and some processes and technologies prevent the mediocre from failing. It is the enlightened leader who can tell which approach is required and then build an environment that maximizes a team’s performance. This book incorporates that ability into the discussion and effectively includes it in the quantitative process in assessing the organization’s productivity potential.”

—Gregory H. Mikkelsen, Integrated Defense Systems, Raytheon
Improving Software Development Productivity
Improving Software Development Productivity

Effective Leadership and Quantitative Methods in Software Management

Randall W. Jensen, Ph.D.
This work is dedicated to my wife, Marge, whose support and patience through this project has been just shy of infinite. And to all the managers and engineers who, for the last 50 years, have contributed so much to my understanding of how the world really works.
## Contents

Preface .......................................................... xv  
About the Author ........................................... xxiii  

Chapter 1: Software Development Issues ......................... 1  
  1.1 Software Crisis ........................................... 2  
  1.2 People Impact on Productivity .......................... 9  
  1.3 Agile Contributions to Development Productivity ...... 13  
  1.4 Magic Bullets ............................................. 15  
  1.5 Development Constraints ............................... 16  

Chapter 2: The Effectiveness Formula ............................ 17  
  2.1 The Formula ................................................ 18  
  2.2 Mechanics of Communication ............................. 19  
      2.2.1 Information Convection ............................ 23  
      2.2.2 Radiation ............................................. 24  
      2.2.3 Four Communication Commandments .............. 24  
  2.3 Culture Issues ........................................... 31  
      2.3.1 Organization Culture .............................. 32  
      2.3.2 Comfort Zone ....................................... 33  
      2.3.3 Confirmation Bias .................................. 34  
      2.3.4 K-12 Phenomenon ................................ 36  

Chapter 3: Importance of Software Management .................. 41  
  3.1 Historical Perspective ................................. 42  
  3.2 Hawthorne Effect ........................................ 45  
  3.3 Theory X/Theory Y ...................................... 47  
  3.4 Agile Software Development ............................ 50  
  3.5 More Real Productivity Improvers ...................... 51  
      3.5.1 Technology .......................................... 52  
      3.5.2 Information Flow in the Development Environment ............................................. 54
## Contents

### Chapter 4: What We Learn from History
- 4.1 Pandora and the Magic Vase ........................................ 58
- 4.2 Productivity Gains from Technology ................................. 59
- 4.3 CMMI .............................................................................. 63
- 4.4 Maximum/Minimum Schedule Is Real ............................... 66
- 4.5 And So ............................................................................ 68

### Chapter 5: Software Development Teams
- 5.1 Software Development Teams ........................................... 72
- 5.2 Team Examples ................................................................ 73
  - 5.2.1 Ad Hoc Team ................................................................. 73
  - 5.2.2 Chief Programmer Team .............................................. 74
  - 5.2.3 Pair-Programming Team ............................................. 75
  - 5.2.4 Cross-Functional Teams ............................................. 81
  - 5.2.5 Tiger Teams ................................................................. 84
- 5.3 Team Positive Effects ....................................................... 86
- 5.4 Team Size ........................................................................ 88
- 5.5 And Then ....................................................................... 91

### Chapter 6: Measuring Organization Capability
- 6.1 The Evolving Concept of Developer Capability .................. 95
- 6.2 The Effectiveness Formula Redux ..................................... 98
  - 6.2.1 Analyst and Programmer Capability ........................... 99
  - 6.2.2 Application Domain Experience ................................. 103
  - 6.2.3 Modern Practices ....................................................... 107
  - 6.2.4 Modern Tools ............................................................. 112
  - 6.2.5 Terminal Response Time ........................................... 114
  - 6.2.6 Hardcopy Turnaround Time ...................................... 114
- 6.3 Organization Capability Rating .......................................... 116
- 6.4 Basic Technology Constant ............................................. 120

### Chapter 7: Faultless Software Corporation Case Study
- 7.1 The Problem .................................................................. 125
- 7.2 Faultless Software, the Corporation .................................. 126
- 7.3 The Software Components .............................................. 128
  - 7.3.1 Mission Planning ........................................................ 128
  - 7.3.2 Command Generation .............................................. 129
  - 7.3.3 Communication .......................................................... 129
- 7.4 Faultless Software Capability Ratings ............................... 130
  - 7.4.1 Analyst and Programmer Capabilities ........................ 130
Chapter 7: Product Complexity ................................. 137
  8.1 A Quantitative Complexity Value ......................... 138
  8.2 Faultless Software Complexity Example .................. 143
    8.2.1 Mission Planning System ............................. 143
    8.2.2 Command Generation System .......................... 144
    8.2.3 Communication System ................................. 144

Chapter 9: Staffing Profiles ................................. 147
  9.1 Effective Use of People ................................... 147
    9.1.1 Lightweight versus Heavyweight Development ....... 151
    9.1.2 Rayleigh Development Model ......................... 152
  9.2 Brooks' Law in Action .................................... 154
    9.2.1 Overstaffing ......................................... 154
    9.2.2 Flat Staffing .......................................... 155
  9.3 A Touch of Reality ......................................... 156

Chapter 10: Seer Software Model Introduction ............... 159
  10.1 Introduction .............................................. 160
  10.2 Software Equation ........................................ 163
    10.2.1 Effective Size ....................................... 168
    10.2.2 Effective Technology Constant ....................... 168
  10.3 Schedule and Cost Estimates ............................. 169

Chapter 11: Development Environment .......................... 173
  11.1 Experience Factors ...................................... 174
    11.1.1 Development System Experience (DEXP) ............ 175
    11.1.2 Programming Language Experience (LEXP) .......... 176
    11.1.3 Practices and Methods Experience (PEXP) .......... 179
    11.1.4 Target System Experience (TEXP) .................... 180
  11.2 Volatility Factors ....................................... 182
    11.2.1 Development System Volatility (DVOL) ............. 182
    11.2.2 Practices and Methods Volatility (PVOL) .......... 182
14.2 Size Uncertainty ........................................ 231
14.3 Source Line of Code (SLOC) ......................... 233
  14.3.1 Executables ........................................ 234
  14.3.2 Data Declarations ................................. 234
  14.3.3 Compiler Directives ............................... 234
  14.3.4 Format Statements ................................. 234
14.4 Effective Source Lines of Code (ESLOC) ............ 235
14.5 Effective Size Equation .............................. 238
  14.5.1 Design Factor .................................... 239
  14.5.2 Implementation Factor ........................... 239
  14.5.3 Test Factor ................................ ...... 239
14.6 Size Growth ........................................... 241
14.7 Size Risk ............................................. 248
14.8 Faultless Software Size Example ...................... 249
  14.8.1 Mission Planning System ........................ 249
  14.8.2 Command Generation ............................ 250
  14.8.3 Communication .............................. .... 251

Chapter 15: Function Point Sizing ......................... 253
15.1 Function Point Counting ................................ 254
15.2 Function Point Components ......................... 254
  15.2.1 Application Boundary ............................ 256
  15.2.2 Internal Logical File ............................ 257
  15.2.3 External Interface File ........................... 260
  15.2.4 External Input .................................... 260
  15.2.5 External Output ................................. 261
  15.2.6 External Inquiry .................................. 261
  15.2.7 Transforms ...................................... 262
  15.2.8 Transitions ...................................... 263
15.3 Unadjusted Function Point Counting ................ 264
15.4 Adjusted Function Points ............................. 265
  15.4.1 Value Adjustment Factor ........................ 266
  15.4.2 Adjusted Function Point Calculation ........... 268
15.5 Backfiring ............................................. 268
15.6 Function Points and Objects ........................ 270
15.7 Zero Function Point Problem ......................... 271

Chapter 16: Maintenance Estimating ....................... 273
16.1 Introduction .......................................... 273
16.2 Software Enhancement ............................... 275
16.3 Knowledge Retention ........................................ 276
16.4 Steady-State Maintenance Effort .......................... 277
16.5 Maintenance Example .................................... 277

Chapter 17: Summary ........................................... 279
17.1 The Effectiveness Formula Revisited. ...................... 279
17.2 People-Process-Project Triad .............................. 281
17.3 The Value of Communication ............................ 283
  17.3.1 Four Communication Commandments ................. 284
  17.3.2 Confirmation Bias ................................... 285
17.4 Management and Motivation .............................. 285
17.5 Teams ...................................................... 287
17.6 Last Thoughts ............................................. 290

Appendix A: Software Estimating Models ..................... 293
A.1 Analogy Models ........................................... 294
A.2 Expert Judgment Models .................................. 295
  A.2.1 Delphi Method ........................................ 296
  A.2.2 Wideband Delphi Method ............................ 297
A.3 Bottom-Up Estimating ..................................... 297
A.4 Parametric Models ......................................... 298
A.5 Evolution of Parametric Software Models ............... 299
  A.5.1 First-Order Models .................................. 300
  A.5.2 Second-Order Models ................................. 302
  A.5.3 Third-Order Models .................................. 304

Appendix B: Additional Reading ............................... 307

Appendix C: Terminology ....................................... 313

Index ............................................................. 327
Preface

Some books are to be tasted, others to be swallowed, and some few to be chewed and digested.

Francis Bacon
Of Studies

Productivity is a measure relating a quantity or quality of output to the inputs required to produce it. Productivity is, in one sense, a measure of efficiency and/or quality. Fortunately, over the past 50 years software development productivity has been consistently measured as the number of delivered source code statements per staff month. The statements are the delivered statements, not the statements written during the development. Source code statements are specified in the programming language used by the software developer. Many statements are written and thrown away during the development process. The delivered statements may include multiple languages such as UML and C++. The statements are counted in the language written, even though a UML statement may contain the power of 40 C++ statements. The input effort is the work required to produce the individual written statement.

My interest in productivity improvement started as early as 1955 while I was an electrical engineering student at Utah State University. I was struggling with a full-time course load in a four-year curriculum, working a part-time job, following my interest in campus politics, and living in a fraternity house. Time management by itself wasn’t adequate to keep up the pace, so I had to somehow reduce the impact of homework and lab reports (increase productivity). I obviously found a simple and logical way to improve my efficiency. Years later I applied the same approach while simultaneously pursuing a Ph.D. in electrical engineering, teaching full time, running a consulting business, and writing two books. I do not consider myself an outstanding intellect, nor an outstanding student. I was guided by the concept that two heads are much more efficient than one when solving problems. This idea led me into a lifelong interest in organization management.

I began experimenting seriously with management technology and its impact on software development costs and schedules during the mid-1970s. Early studies led me to believe that the people aspects of software development had significant impacts on both software productivity and quality. My first serious
experiment with software development teams began in 1975 and yielded a 175 percent increase in productivity and a nearly three-orders-of-magnitude decrease in errors. I have been told by several colleagues that the 1975 experiment was the first documented use of pair programming.

Chuck Tonies and I wrote in our 1979 text *Software Engineering*\(^1\) that the software engineer’s value \(V\) to an organization is dependent on three attributes of the Effectiveness Formula: communications skills \(C\), management concept awareness \(M\), and technical ability \(T\), or

\[
V = \frac{C}{M(T)}
\]

The Effectiveness Formula is implicitly fundamental to the Agile development approaches that have become popular today. The formula is also fundamental to the quality and productivity improvements in the traditional development approaches as well. People, motivation, and communications are key attributes of all successful projects.

The purpose of this text is to explore the concepts that are the primary factors that drive a productive environment, provide the means of evaluating the effectiveness of an organization’s development environment, and project the productivity impacts of decisions made by managers during the inception and execution of their software development projects.

The productivity impacts go far beyond those associated with technology changes that have been the norm over the past several decades and include the more dramatic impacts that are part of the management and communication attributes of the Effectiveness Formula.

I will explore the makeup of the Effectiveness Value ratings and the means of improving these ratings in an organization while simultaneously improving the organization’s productivity, effectiveness, and estimating ability; equally as important, I will give you a better understanding of the development process and the interaction between management style and the environment so that you can significantly improve development productivity. Instead of using the environment parameters to calculate development costs and schedules, the metrics can be used to gauge the impact of any management decision that affects the environment.

The text can help answer questions about the project environment, such as “What does the use of cubicles cost the project?”, “What is the cost of changing the programming language of choice from Ada to C++?”, or “How much will productivity improve by increasing my Capability Maturity Rating (CMMI) from Level 3 to Level 5?”

---

A second principal objective of this text is to establish an estimating methodology that can, in the hands of a trained analyst, produce realistic estimates of the schedule and resources required for software development under a broad variety of project and environmental conditions.

The obvious, most likely audience for the text is software developers. Software developers include all managers and professionals interested in productivity improvement. The concepts described here apply equally to both Agile and traditional software developments. The use of communications, teamwork, and the environment are implicit in pair programming as an example of Agile development.

Software development is the ideal vehicle to use for explaining the productivity concepts because of the wealth of documentation and historical data spanning over 50 years.

While I was trying to compile a list of the audience members who would benefit from the material presented here, I thought back to the first beneficiary, a struggling student trying to complete his college career in electrical engineering with the parallel need to work part time. All the concepts about communications, teamwork, and environment discussed in this text were part of making that career possible. With that in mind I expanded the list of beneficiaries of the ideas in this material to include not only software managers, engineers, and programmers, but also those in any discipline (information technology, manufacturing, communications, education, etc.) that involves people and the effective, efficient use of those people. The famous Hawthorne Experiment in the early twentieth century applied these concepts to equipment manufacturing with great success. The ideas are truly universal.

Software development is a people-centered process, as described by DeMarco and Lister in their book *Peopleware*, not the traditional technology-centered process that has been popular since the 1950s. Although the traditional development process is still the major process used in large organizations today, *Improving Software Development Productivity* provides a road map that can quantitatively support the transition from traditional to modern development styles and environments. Concepts of total quality management (TQM), including the use of software development teams and Theory Y management, become obvious management technologies to be considered if more substantial gains in productivity and quality are to be realized. The transitions, however, are not all milk and honey. There are short-term penalties to be paid for any process change, whether it is the latest state-of-the-art CASE tool, a new programming language, CMMI, or a development team approach.

My work in the field of software cost and schedule estimation began in 1978 while supporting a proposal effort for a large space-based software system. I was tasked by the Hughes Aircraft Company Space and Communication Group

---

(my employer) to develop a simulation model that would be the basis of their software development estimates. I derived a mathematic model that produced realistic estimates through the use of environment parameters including both developer capability and project-imposed constraints. The completed software development project data, accumulated from multiple sources that date from as far back as 1960, helped create the mathematic model and the organization Capability Calculator used in this text. It is amazing, maybe even frightening, that the model formulated in 1980 still produces realistic effort and schedule estimates independent of the development approaches today.

Much of this text explores this simple concept and its importance to software development productivity. Simply stated, the productivity of a development environment and the resulting software development cost and schedule are driven by the product of three important attributes: communications, management, and technology. Although this text focuses on software, the principles can also be applied to other fields without modification.

Improving Software Development Productivity: Effective Leadership and Quantitative Methods in Software Management is comprised of two major parts: Effective Leadership and Quantitative Methods.

---

Effective Leadership

The first seven chapters focus on effective leadership and measurement of organizational capability.

- **Chapter 1, Software Development Issues.** This chapter discusses the “software crisis” and the technology approaches that have been used since the 1960s to resolve the crisis issues. Productivity has improved very slowly from 1970 to the present in spite of great improvements in tools, languages, and development approaches.

- **Chapter 2, The Effectiveness Formula.** This chapter discusses the importance of the three attributes of the Effectiveness Formula—communications, management, and technology—in productivity improvement. The mechanics of effective communications and some important cultural issues that impede effective development improvement are also discussed.

- **Chapter 3, Importance of Software Management.** This chapter discusses two people management principles in effective development: the Hawthorne Effect, and Theory X/Theory Y management principles. These

---

concepts are the basis of modern people management. The relationship between Agile software development and these principles is discussed as an example.

- **Chapter 4, What We Learn from History.** This chapter describes what we have learned from history about software development productivity, technology productivity contributions, the productivity impact of CMMI, and the result of optimistic pricing and scheduling.

- **Chapter 5, Software Development Teams.** Software development teams are explored in this chapter. The good, the bad, and the ugly, and their impacts on development productivity, are discussed.

- **Chapter 6, Measuring Organization Capability.** A process for measuring software development productivity is presented that can be applied to your development organization to reveal its inherent capabilities, basic technology constant, and the relative standing of your organization with respect to industry. A tool to support the evaluation is contained in the chapter.

- **Chapter 7, Faultless Software Corporation Case Study.** This chapter presents a case study in capability assessment that summarizes the material in the first section of the text.

---

**Quantitative Software Development Management**

The remainder of the text discusses the application of quantitative management in the delivery of software products within cost and schedule constraints.

- **Chapter 8, Product Complexity.** This chapter discusses the impact of software system complexity on software development cost and schedule constraints.

- **Chapter 9, Staffing Profiles.** This chapter presents an evaluation of optimum development staffing profiles for software development. Optimum staffing is a function of product complexity and independent of the development approach.

- **Chapter 10, Seer Software Model Introduction.** This chapter presents an introduction to the Jensen software model used for quantitative development management (the basis for the SEER-SEM and Sage estimating tools).

- **Chapter 11, Development Environment.** This chapter discusses the impact of the development environment on product cost and schedule. The environment evaluation includes considerations related to experience, volatility, and other management constraints.
• Chapter 12, Product Characteristics. This chapter discusses the impact of constraints imposed by the product characteristics and requirements on productivity and the software development environment.

• Chapter 13, Development Schedule and Cost Estimates. This chapter uses the Faultless Software Corporation case study to describe the estimation process on the software development cost and schedule. The estimates are limited by the constraints imposed by the organization capability and environment.

• Chapter 14, Effective Size Estimation. Effective size is not simply an estimate of the number of new and modified software lines of code. This chapter explores the effective size needed to predict the estimate development cost and schedule.

• Chapter 15, Function Point Sizing. This chapter contains an introduction to function point sizing of traditional software products. The use of object points is also discussed as an alternate sizing method for object-based development.

• Chapter 16, Maintenance Estimating. Maintenance estimating is more than simple software enhancements estimates. The impact of knowledge retention on software maintenance is discussed as a major addition to software product support.

• Chapter 17, Summary. This chapter reviews the information presented in the first 16 chapters and applies the important concepts to nonsoftware development environments.

• Appendix A, Software Estimating Models. This appendix discusses the evolution of quantitative software estimating models and the capabilities of each of the major estimating approaches.

• Appendix B, Additional Reading. This appendix contains a broad list of additional reading that covers the effective leadership topics (communications, people management styles and issues, and technology) as well as quantitative management and estimating topics.

• Appendix C, Terminology. This appendix contains common definitions of terms used in the text.

• Capability Calculator Access. The Capability Calculator spreadsheet used throughout the text to support the organization capability calculations and software effort and schedule estimates can be freely downloaded from the Prentice Hall website for the book, www.informit.com/title/9780133562675; just click on the “Downloads” tab.
Acknowledgments

There are several individuals whom I wish to acknowledge in the development of this material. First and foremost are the contributions of Chuck Tonies and Ken Hubbard of Hughes Aircraft’s Space and Communication Group in supporting me in the development of the Jensen model and encouraging me to pursue the experiments that became the foundation of the management concepts discussed here. I cannot forget project manager Frank Wolfe for his active support in the estimating activity, and SCG group executive Don Forster for granting me ownership of the estimating technology and encouraging me to commercialize the Jensen model and launch an estimating consulting career. Another key person who worked with me through the years in pursuing the Jensen model and many estimating ventures is Suzanne Lucas, a long-time friend and colleague.

There are four engineers from the USAF Software Technology Support Center at Hill AFB who were part of the software estimating team and contributors and valued collaborators for all the estimating, training, and technology extensions during my time at STSC: Les Dupaix, Mark Woolsey, Thom Rogers, and Brent Baxter.

Finally, I don’t want to forget my wife of many years and many textbooks, Marge, for all her support in this endeavor. This one only took about 25 years to finish.

—Randall W. Jensen
This page intentionally left blank
About the Author

**Randall W. Jensen** first performed productivity improvement experiments as a freshman EE student in 1955. He went on to earn his Ph.D. and worked to find ways to improve productivity in the Hughes Aircraft Company’s Space and Communications Group. Simultaneously, he was asked to develop a computer model to estimate the cost and schedule for large software development projects. The resulting computer model provided a tool that development managers can use to measure organization capability and quantitatively predict the productivity impact of management decisions related to personnel, management style, and the development environment. The key productivity attributes—which he and Chuck Tonies published in their 1979 Prentice Hall book, *Software Engineering*—were communications, management, and technology. His research into productivity since then led in 1990 to an improved computer-based estimating model, based on the three attributes and this text.
This page intentionally left blank
Chapter 1

Software Development Issues

“When I use a word,” Humpty Dumpty said in a rather scornful tone, “it means just what I choose it to mean—neither more nor less.”

Lewis Carroll

Through the Looking Glass

This introductory chapter highlights some of the software issues that have been around since the 1960s and are still prevalent today. Whether they have been ignored because they are not publicized, or because we do not want them to cloud our thinking, history is history.

The first question that I hope entered your mind when you opened this book is about what you can do to improve software development productivity in your organization. You aren’t the first person in the past 40 years or so to ask this question. In spite of all the attempts that have been made using similar approaches, they have resulted in limited results in real productivity gains.

I have been trying to wrap my arms around the software productivity problem since the 1970s, when I was formally tasked by my manager to find a way to improve software development productivity in his organization. Because I was new to the organization, I was not bound to consider only potential solutions that were consistent with the standards and culture of the organization. I was able to observe as an outsider the normal development approaches and tools used in that development environment.

I had some early successes in the productivity improvement task that have guided my productivity research since that time, even though most of the task successes were abandoned by the organization because they were not consistent with the organization’s culture.

I also made some of the same errors you have probably made on your way to reading this book. I tried most of the new technology solutions with the hope that each one would have a marked, positive impact on development
productivity. One of the first technologies was Programmer’s Workbench (PWB), which made all the current and previous versions of the software code accessible, reviewable, and testable with the idea that it was going to reduce errors and greatly improve the product. PWB worked and removed some of the product problems, but it didn’t really improve productivity. Our organization adopted PWB as a development standard, hoping that the quality and productivity improvements would be a good financial investment. Quality and the development process did improve due to the more efficient handling of the source code and the ability to quickly repair errors. Quick corrections also made it possible to make changes without much thought. Development productivity only improved marginally.

Much of the development work I was involved with during my career included real-time software system development. The techniques I learned using the Hatley/Pirbhai real-time systems specification methods contributed to the quality of that work in a very positive way. However, it didn’t significantly improve productivity even though the resulting designs were better.

I began collecting data in the mid-1970s to investigate the impact of the environment on software development cost and schedule. The environment initially included the organization facilities, tools, and processes, but as time passed, it also began to seriously include organization management and culture. As the quantity and quality of the data improved, the data became the foundation of a cost and schedule estimating model that highlighted the productivity improvement areas of concern. Note: In spite of the technology focus of most publications and the belief of many managers, technology offered the smallest productivity payoff of the development environment elements.

I ultimately constructed a model that will be used throughout this book to help explain the productivity impacts of the elements in the development environment and the decisions you make as a manager relative to your environment.

1.1 Software Crisis

The term “software crisis” refers to a set of problems that highlight the need for changes in our existing approaches to software development. The term originated in the late 1960s about the time of the 1968 NATO Conference on Software Engineering.1 At this conference, a list of software problems was presented as the major development concerns. The problem list included software that was

1.1 Software Crisis

- Unreliable
- Delivered late
- Prohibitive in terms of modification costs
- Impossible to maintain
- Performing at an inadequate level
- Exceeding budget costs

By the way, this list of problems still persists in much of the software development industry today, some 40 years later. The list has been reduced in many organizations, but when we look at the individual problems in the list, we can observe a common thread—a lack of a realistic schedule (delivered late).

A notion pervading the conference was that we can engineer ourselves out of any problem. Hence, the term “software engineering” was coined. Software engineering had to be a potential solution to the problems, but we have to look at what happened after the term was coined. One of the significant conference outputs was a software engineering college curriculum. However, the curriculum that was produced just happened to be identical to the computer science curriculum of that day. Changing the subject name from “computer science” to “software engineering” accomplished very little.

Crisis is a strong word. It suggests a situation that demands resolution. The conditions that represent the crisis will be altered, either toward favorable relief or toward a potential disaster. According to Webster’s definition, a crisis is “a crucial or decisive point or situation.” A heart attack is a crisis where we either live or die. By now, the crisis should have been resolved one way or another. In retrospect, the term exigence fits the situation better than crisis because there is no discernible point of change for better or worse. A skin rash is an exigency.

Looking a little deeper into the list of problems, we find that the perceived solution to the software development problems was technology. According to the results in Figure 1.1 from the 2013 Standish Chaos Manifesto, technology has not been the total solution to project success.

The Chaos report divides projects into three classes: successful, challenged, and failed. Only about 29 percent of the 2004 projects evaluated in the 2004 study were classified as successful. Fifty-three percent were delivered but with significant overruns in cost and schedule while delivering an average of only 64 percent of the features of the original requirements (challenged). The overruns

---

2. Exigence (or exigency): The state of being urgent or pressing; urgent demand; urgency; a pressing necessity.
averaged about 84 percent in schedule and 56 percent in cost. The remaining 18 percent were cancelled before delivery (failed).

About 39 percent of the 2012 projects evaluated were successful. Forty-three percent were delivered, but with significant average overruns of nearly 59 percent of cost and 74 percent of schedule while delivering only 69 percent of the original requirements (challenged). Still, about 18 percent were cancelled before delivery (failed).

We can observe from this report a gain of successful project completions of about 10 percent during the past decade due to shifts in the development environment, including better schedule and cost estimates, processes, technology, and team performances. Most projects have problems, but more often they are people problems related to culture rather than technological problems.

A second study shown in Figure 1.2, which encompasses a large number of major aerospace (ground and space) software projects, illustrates a relationship between system size and development time. Three things are evident from this historic data. First, there is an apparent maximum size of 200,000 source lines for projects that are completed and delivered.

Second, there is also an apparent maximum schedule for the development of software system components. There is little data in this set that required more than four years to complete.

Third, the report indirectly contained two important pieces of information related to development productivity. If a development project lasted more than five years, it was outdated and no longer useful. With projects including more than 200,000 source lines, the number of people required on the development team overwhelmed the team’s ability to produce the product.

There is a close relationship between productivity and software estimating tools. Productivity achieved on the last development project is close to the

---

productivity that will be used to determine the next project’s cost and schedule. Also, the parameters used by the tools are indicators that can, or should, be used to determine management actions to improve software development productivity. The standard metric for software development productivity has been the number of delivered source lines of code, independent of the development language per person month of effort since the beginning of project history recording.

One consistency that has aided the developers of software estimating tools is the use of a delivered source line of code as a measure of the software product size. That isn’t a perfect measure, because it doesn’t accurately account for rework, but it has been used since the 1960s and illustrates the general productivity trends that we observe.

The software estimating tools in widespread use today evolved from models developed in the late 1970s to early 1980s using historical project data available at the time. The widely used tools today include COCOMO II, Price-S, Sage, and SEER-SEM. It is important to note that these mature tools are as useful today as they were 30 years ago when they were first formulated. Input data parameter sets (analyst and programmer capability, application experience, use of modern practices and tools, etc.) developed for Seer and COCOMO to describe organizations in the early 1980s are, oddly enough, still accurate and

---

**Figure 1.2** 1996 Aerospace Software project completion study

---

applicable today. The organization parameter definitions have changed very little, and the values of those parameters haven’t changed much, either. Fortunately for the estimating model developer, the traditional software development culture has changed very slowly. Agile software development has introduced a major cultural shift that has already led to a new way of thinking about efficient development.

There have been several development technology breakthroughs during the past 40 years that have significantly decreased the cost of software products. For example, the introduction of FORTRAN and COBOL decreased the cost of a given product functionality to one-third of the cost achievable when implemented in Assembler due to the decrease in the source lines of code required to achieve the product functionality. The transitions from C++ to the newer visual languages and the advent of object-oriented structures created significant software cost savings, because the required number of source lines have continually decreased. However, when we look at the effort required to produce a single line of source code in any given programming language (old or new), we see that traditional software development productivity (measured from start of development through delivery or software-system integration) has increased, with little blips and dips, almost linearly at the rate of less than two source lines of code per person month (SLOC/PM) per year, as shown in Figure 1.3.

We have learned new things about software development during this period. The development environment focus was almost entirely on the product during the 1960s and early 1970s. The principle activity once the requirements were established was programming, or should I say coding. Programmers were simply programmers. Software development technology, namely programming languages, improved as the system requirements grew to manage the size and

![Figure 1.3](image-url)  
*Figure 1.3  Traditional software development productivity gains—1960 to 1990*
complexity of the tasks increased. Development platforms improved to support the ever-increasing size of software systems.

The first major software system I encountered in my career was a real-time airborne weapons system with approximately 100,000 delivered source lines of assembler code. The development was started in the early 1960s and delivered almost three years later. The system included both radar and weapons software. Today that system size would not be memorable, except for several constraints that we do not have today. First, the software amounted to 50 boxes of punched cards implementing a single component. System development processes and standards did not exist. Modern software methods beginning with structured programming were not created until years later. There were no tools to manage source code or other development and test products. Documentation was created with a typewriter. The development team was approximately 35 engineers, of whom 20 were referred to as programmers. Much of the work was performed in a lab environment with the hardware engineers. This project achieved a productivity of a little over 70 SLOC/PM.

Ten years later, software systems took on a different character than the real-time software systems of the 1960s. Development and target computers were much larger and faster. Programmers had become software engineers. The major third-generation programming languages were FORTRAN, COBOL, PL/I, PASCAL, and C. Keypunches had become digital files, even though the DEC Programmer's Workbench included just experimental development environments. Development standards were still in their infancy, but their necessity was obvious. Structured programming was an important new development strategy. Project management was becoming a major development factor.

Had third-generation languages been available and capable of implementing the airborne weapons system just described in 1960, the size could have been reduced to 33,000 source lines, and the product could have been delivered a year earlier using a staff of about 25 people. The improved schedule and the decreased cost were the results of the reduced size.

Programming languages are included in one branch of technology that has continued to change over time to keep the effective product size at a manageable level. From the first assembler programming language to the third-generation languages, there was a 3:1 reduction in program size. The object-based languages such as Visual C and Visual Basic created the ability to build larger-scale objects with a further reduction in size. The current use of Universal Modeling Language (UML) and state charts to automatically generate C++ code (auto-code generators) projects a 40:1 size improvement over C++. We still measure productivity based on the written source code for the project.

1960 to 1990 represented a historical period in which software development technology improved dramatically. Alvin Toffler described the phenomenon in
As soon as we started programming, we found to our surprise that it wasn’t as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.

Size and complexity brought about the need for processes to manage the software development. The early waterfall process of requirements analysis, software design, code, test, and integration became the development standard.

Technology improved along the line shown in Figure 1.3. Each new technology change offered new and often better approaches to the complexity of the software development process. New approaches to the development process such as the software spiral method proposed by Boehm\textsuperscript{12} and the

\begin{flushleft}


\end{flushleft}
Gilb\textsuperscript{13} evolutionary delivery method provided ways to manage requirements complexity and volatility. Traditional software development productivity increased with the new technologies, but at only about 1.5 delivered SLOC/PM per year improvement.

\section*{1.2 People Impact on Productivity}

All organizations and development methods can be represented by the People-Process-Project triad shown in Figure 1.4. The projects vary from industry to industry with the products of each industry contained in the Project terminal. All organizations contain a People terminal representing the physical environments the people are part of and the management environments of those individuals. The Process terminal in the triad represents the development processes used by the people to produce the organization’s products. Organizational capability and development productivity are largely driven by the communication and management attributes of the People terminal in the triad model.

The Process terminal determines the process used to develop the organization products. For the purpose of this text I am going to divide the processes into two camps. The first camp I will refer to as the “traditional” processes. The traditional processes include the classic waterfall and spiral\textsuperscript{14} development and other variations of the waterfall process. The core of the spiral is also a waterfall process. The second camp, which I will refer to as “Agile,” comprises the varied Agile software development processes. Agile methodologies include pair programming\textsuperscript{15} (1975), Scrum\textsuperscript{16} (1995), Crystal Clear\textsuperscript{17} (1996), Extreme Programming\textsuperscript{18} (1996), Adaptive Software Development,\textsuperscript{19} and the Dynamic Systems Development Method (DSDM)\textsuperscript{20} (1995) among others.

\begin{itemize}
  \item \textsuperscript{17}Cockburn, A., \textit{Crystal Clear: A Human-Powered Methodology for Small Teams} (Boston, MA: Addison-Wesley, 2004).
  \item \textsuperscript{18}Beck, K., \textit{Extreme Programming Explained} (Boston, MA: Addison-Wesley, 2001).
  \item \textsuperscript{19}Highsmith, J., \textit{Adaptive Software Development} (New York, NY: Dorset House, 2000).
  \item \textsuperscript{20}McCarthy, J., \textit{DSDM Dynamic Systems Development Method} (Redmond, WA: Microsoft Press, 1995).
\end{itemize}
This text is not going to describe in detail or advocate any of the methods in either of the two camps, but concentrates on the People terminal and its impact on software development productivity.

Until the mid-1970s people were the most ignored part of the triad. The People terminal is common to both the traditional and Agile process camps. For example, pair programming, an Agile method, is very dependent on communications and management support to function, and the pair programming concept works well in the traditional environment.

I have separated the People terminal and added the communications and management connectors to emphasize their significance in the model. Barry Boehm wrote in 1981:

> Poor management can increase software costs more rapidly than any other factor. Each of the following mismanagement actions has often been responsible for doubling software development costs. . . .


Of course, you have to read the first 485 pages of his book to find this simple yet profound statement. Most readers don’t seem to read that far. Weinberg’s Second Law of Consulting added a supporting observation:

> No matter how it looks at first, it’s always a people problem.


1.2 People Impact on Productivity

Boehm isolated a group of 16 “cost drivers” that determined the cost impact of each driver on the software product. Weinberg grouped the cost drivers into four categories—tools, people, systems, and management—according to the Boehm cost-driver definitions. The results, which should point to the group with the highest payoff, are plotted in Figure 1.5 according to the percentage of total cost.

Figure 1.6 illustrates the vigor with which the Software Engineering Institute research has pursued a technology solution (a silver bullet) to the productivity problem, according to the number of papers published in the areas defined by Weinberg between 1986 and 1991. The key to increased productivity doesn’t appear to be technology.

Weinberg demonstrates the results of this research by comparing the relative percentages of Software Engineering Institute publications in major activity areas of technology (tools), people (education), systems (development

---


environments), and management with the relative productivity gain for each group. According to Weinberg, the most significant productivity improvement area is, by far, the management activity area.

Barry Boehm argued, “Poor management can increase software costs more rapidly than any other factor.” But he explains his omission of management as a cost driver in the following:  

Despite this variation, COCOMO does not include a factor for management quality, but instead provides estimates which assume that the project will be well-managed.

The well-managed concept does not work in this context. Boehm’s analyst and programmer capability-rating definitions evaluate the environment independent of management. Without management factors, we cannot distinguish between well-managed and poorly managed projects, nor can we consider the implication of communications and organizational culture on productivity.

The software management problem, including cost and schedule estimating and their optimistic approaches, can be described by the following statement made by Fred Brooks in *The Mythical Man-Month* (1975), which sums up some of the major difficulties of traditional project management:

More software projects have gone awry for lack of calendar time than for all other causes combined. Why is this cause of disaster so common?

First, our techniques of estimating are poorly developed. More seriously, they reflect an unvoiced assumption which is quite untrue, i.e., that all will go well.

Second, our estimating techniques fallaciously confuse effort with progress, hiding the assumption that men and months are interchangeable.

Third, because we are uncertain of our estimates, software managers often lack the courteous stubbornness of Antoine’s chef.

Fourth, schedule progress is poorly monitored. Techniques proven and routine in other engineering disciplines are considered radical innovations in software engineering.

Fifth, when schedule slippage is recognized, the natural (and traditional) response is to add manpower. “Like dousing a fire with gasoline, this makes matters worse, much worse. More fire requires more gasoline and thus begins a regenerative cycle that ends in disaster.”

---


The fifth cause listed by Brooks is almost universally experienced today (almost 40 years later) in traditional software development. It is obviously a bad approach to solving the schedule slippage problem, but it is normally the first solution choice applied without reasonable judgment. At least cost and schedule estimating is much better than at the time of Brooks’ assessment. Optimism is still widely used as a management tool.

1.3 Agile Contributions to Development Productivity

In 1975 a two-person programming team experiment applied the triad software development model to the implementation of a real-time software system executive. The process followed was a modified waterfall approach replacing the traditional programmer with a two-person team using a single workstation; one programmer “driving” and the second programmer “observing, reviewing, or navigating.” The navigator is not an idle participant. In a military sense, the navigator reviews the implementation in real time and focuses on the “strategic” direction of the work by recommending ideas for improvements and pointing out potential problems to address for the future. The driver focuses on the “tactical” aspects of completing the current task, using the navigator as a safety net and guide. (The details of the experiment are described in Chapter 5.)

This team approach became known as pair programming several years later. The focus of the experiment was the impact of tight communications and modern management techniques (not processes) on software development productivity. The very positive results of the experiment led to the Software Effectiveness Formula discussed in Chapter 2, and it led to the definition of the development organization capability model (Chapter 6) used in software effort and schedule estimation tools today.

Pair programming is considered to be an Agile software development technique even though this technique is often used in a traditional waterfall development approach. The productivity gains from pair programming are very beneficial in either approach.

Agile software development in general includes a group of software development methods based on iterative and incremental development, where requirements and solutions evolve through collaboration between self-organizing, cross-functional teams.

Incremental software development has existed throughout the history of software. It existed before the introduction of any formal development approaches. I have heard some old-timers refer to it as “programming when it was
real programming.” Agile software development methods evolved in the mid-1990s as a clear alternative to the formal waterfall development approaches of the 1970s.

Agile development promotes adaptive planning, evolutionary development, and delivery, and it encourages rapid and flexible response to change. It is a conceptual framework that promotes foreseen tight interactions throughout the development cycle.

I am not going to take sides in the continuing debate about the merits of either the formal waterfall approaches (including evolutionary spiral development), or the iterative Agile approaches to development and delivery, or put myself somewhere in between. In terms of improving software development productivity the Agile development methods offer two clear benefits up front:

1. The use of software development teams, and
2. The effective use of communications.

Both of these benefits are congruent with the attributes of the Effectiveness Formula mentioned earlier. Chapter 2 will be devoted to the implications of this formula.

It is appropriate here to consider the impact of the two benefits. Alistair Cockburn uses a term called *virtual teams* in his book discussing Agile software development. His definition of virtual does not include sitting together. In the traditional process world, all teams are virtual; that is, there is no requirement for co-located teams or communications. Of course virtual team members do not have to sit together, so we can ignore the fact that most team members are isolated in individual boxes. Boxes are communications barriers. Productivity is related to the speed of development, or as Cockburn puts it, the speed of development is directly related to the time and energy cost per idea transfer. As the distance between the team members increases, so does the cost per transfer.

Extending the cost-per-idea transfer idea to development decisions, the time and energy cost per decision also affects the productivity of the project. The separation between the manager and the source (the discoverer of the problem) directly impacts the project. The time and cost to adjust the project to the decision is also important in productivity.

1.4 Magic Bullets

Managing, as well as estimating, is magic for most estimators and managers. Well-known science fiction author Arthur C. Clarke’s Third Law\(^{29}\) states:

> Any sufficiently advanced technology is indistinguishable from magic.

This illustrates one of the primary problems with software management today. The “magic” creates an unreasonable trust in the development technology and environment and a lack of rational thought, logical or otherwise.

With magic we expect the impossible, and so it is with management and estimating as well. When something is magic, we don’t expect it to follow logic, and we don’t have to apply our common sense. When the estimate is not the cost and schedule we want, we can simply change the inputs to the cost/schedule algorithm in the estimating tool and produce the estimate we desire. That is why so many projects overrun, and we consistently blame the failure on the projects, not the estimates. The same is true with development management. When the last development had a productivity of 100 SLOC/PM, and the proposed project has a predicted productivity of 150 SLOC/PM, we must ask ourselves what changes were made to the environment to achieve productivity improvement. The common answer I have heard during early project reviews is, “We are going to work smarter this time.” That response doesn’t change the people, the environment, or the management approach, and the proposed gain will not materialize. Doubling the engineers’ salaries won’t make the gain happen either.

Several cost and schedule estimation methods have been proposed over the past 25 years with mixed success due in part to limitations of the estimation models. A significant part of the estimate failures can be attributed to a lack of understanding of the software development environment and the impact of that environment on the development schedule and cost.\(^{30}\) The physical and management environment imposed by the project manager are major drivers in the software development.


1.5 Development Constraints

There is a model of software development as seen from the project-control point of view. This model has only four variables:

- Cost
- Schedule
- Quality
- Scope

The shareholders (users, customers, etc.—all external to the development) are allowed to set three of the four variables; the value of the fourth variable will be determined by the other three.

Some managers attempt to constrain all four variables, which is not possible unless the constraints can all be achieved without penalizing the other constraints. When one attempts to set all four, the first visible failure is a decrease in product quality. Cost and schedule will then increase in spite of our most determined efforts to control them. If we choose to control cost and schedule, quality and/or scope become dependent variables.

The values for these attributes cannot be set arbitrarily. For any given project, the range of each value is constrained. If any one of the values is outside the reasonable range, the project is out of control. For example, if the scope (size) is fixed, there is a minimum development time that must be satisfied to maintain that scope. Increasing funding (increasing the development staff) to decrease the minimum development time further will actually increase the development time while increasing the project cost, because of the increased communication and management load.

Software management/estimating tools allow us to make the four variables more visible so that we can compare the result of controlling any or all of the four variables (look at them as constraints) and their effect on the product.

Greg Mikkelsen, a former colleague from Raytheon Corporation, told me, “When it comes to definitions we rarely differentiate between management and leadership. Managers follow a set of principles and processes and report on results, and leaders take those principles and processes and motivate for excellence. Often the difference between successful and unsuccessful projects (unprecedented or not) is the type of management/leaders a project has.” I like to differentiate managers and leaders as sheepherders and shepherds, respectively. I am going to devote much of the following chapters to Mikkelsen’s rational observation.
Index

1975 pair programming experiment, 77–81
A priori learning curve cost, 105, 175, 178
AAF (Adaptation Adjustment Factor), 237–238, 313
ACAP (Analyst Capability) rating
  FSC case study, 135
  organization capability, 101–102, 118–120
ACT (Annual Change Traffic), 275
Action Office, 27–28
Ad Hoc teams, 73–74
Ada programming language
  cross-functional team study, 83–84, 86–87
  expansion factor, 210
  expectations, 35
  introduction of, 61–62
  mastery of, 177–179
  Mission Planning task, 211–213
Adaptation Adjustment Factor (AAF), 237–238, 313
Adaptive maintenance, 274
Adjusted function point (AFP) count, 254, 265–266
defined, 313
  value adjustment factor, 266–268
Aerospace Corporation study, 4, 66–68, 167
AEXP (Application Domain Experience) rating, 175
  as cost driver, 224
  FSC case study, 131–132, 135
  Mission Planning task, 210
  organization capability, 103–107, 118–120
AFP (adjusted function point) count, 254, 265–266
defined, 313
  value adjustment factor, 266–268
Agile development
  cross-functional teams, 81, 84
  cultural shift from, 6, 8
defined, 313
  iterations, 206
management and people focus, 45, 288
  overview, 50–51
  pair programming, 75–76, 289
  People-Process-Project triad, 9–10
  practices, 108
  productivity contributions, 13–14
  staffing profiles, 151–152
  technology constants, 121, 124
  tiger teams, 85
Agile Manifesto, 50, 72, 75, 288
Aha moments, 37
Albrecht, A. J., 227
Algorithmic estimating models, 294
Algorithms, 313
Analogy software estimating models, 294–295
Analyst capabilities
  FSC case study, 130–131
  overview, 99–103
Annual Change Traffic (ACT), 275
Application boundaries with function points, 254, 256–257
Application Domain Experience (AEXP) rating, 175
  as cost driver, 224
  FSC case study, 131–132, 135
  Mission Planning task, 210
  organization capability, 103–107, 118–120
Application programs, 313, 321
Arbiter, Petronius, 71
Architecture design, 313
Assembly language code
  costs, 6
defined, 314
  program size, 7
  transition from, 59–61
Atmosphere for teams, 87, 289
Attitude changes, 46
Augustine, Norman, 227, 293
Aural portion in communication, 22
Autonomy in cross-functional teams, 81
Backfiring process
 defined, 314
 function points, 268–270
Bacon, Francis, 35
Baker, Terry, 74
Barriers to communication, 14, 23–25
Baseline effective size value, 248
Basic Operating Rules, 27, 83
Basic Technology Constant, 33
 defined, 314
 overview, 120–124
 projection, 201–203
 relation to Organization Capability Rating, 120, 123–124
 variations, 208–209
Beck, Kent, 76, 90, 289
Behavior
 real-time operation, 194–195
 Theory X/Theory Y, 47–49
Behavioral schools of management, 43
Bias, confirmation
 as communication barrier, 285
 defined, 316
 impact, 34–36
 for structured programming, 34–35, 60
Bits, 314
Black box elements, 229–230
Body language in communication, 21
Booch, Grady, 270
Bottom-up software estimating model,
 297–298
Brainstorming
 pair programming, 77
 software development teams, 86, 288
Brooks, Fred P., 139–140
 complexity, 173
 flat staffing, 155–156
 management issues, 12–13
 overstaffing, 12, 74, 137, 139, 147, 154–158
Business application function points, 258
Bytes, 314
C++ programming language
 Command Generation task, 216
 mastery of, 177–179
 Mission Planning task, 211
Capability
 COCOMO definition, 95
 defined, 93, 119, 314
 evolution of, 95, 117
 measuring. See Measuring organization capability
 organization rating related to development productivity, 93–94
 Seer definition, 96–97
 Capability Calculator tool
 Command Generation task, 217
 communication task, 220
downloading, 119
for schedules, 169–170
Capability Maturity Model (CMM) concept,
 45, 65
Capability Maturity Model Integration (CMMI)
 defined, 314
 FSC case study, 133
 levels in modern practices, 109
 overview, 63–66
CARD (Cost Analysis Requirements Description), 231–232
Carroll, Lewis, 1, 291
Case study. See Faultless Software Corporation (FSC) case study
Centralized multiuser systems in DEXP, 175, 180–181
CER (cost-estimating relationships), 298
Changes
 attitudinal, 46
 Command Generation task, 216
 Communication task, 219
 Mission Planning task, 211
 Rehosting requirement (HOST) rating, 191–192
 Chaos and tiger teams, 147–148
 CHAOS report, 3–4
 Chief programmer teams
 defined, 314
 overview, 74–75
Clarke, Arthur C., 15
CMM (Capability Maturity Model), 45, 63
CMMI (Capability Maturity Model
Integration) defined, 314
FSC case study, 133
levels in modern practices, 109
overview, 63–66
Co-location of teams, 89
Co-pilots in pair programming, 78
Cockburn, Alistair, 14, 23, 25
COCOMO tools
AAF in, 237–238
capability definitions, 95
Doty Associates model influence, 160–161,
300
management quality factor in, 12
models, 206
usefulness, 5
Code and unit tests, 314
Coding War Games, 79
Cohesion, 315
Collaboration. See also Communication
K-12 phenomenon, 36–39
vs. noise, 25
College engineering graduates useful life
expectancy, 8
Comfort zones
defined, 315
overview, 33–34
pair programming, 80
Command Generation task
development environment, 216–217
effective size estimation, 215, 250–251
FSC case study, 129, 144, 215–218
product characteristics, 218
Commanding, software management for, 42
COMMENT statements, 234
Commercial Off-the-Shelf (COTS) products
defined, 316
size impact, 231
Common area in pair programming, 80
Common Criteria, SECR (Security
Requirements) based on, 198
Communication
Agile software development, 51
barriers, 14, 23–25
commandments, 24–31, 284
and complexity, 140
confirmation bias in, 285
development environment for, 219–220
effective size estimation, 251–252
Effectiveness Formula, 18
estimate for, 221
and experience, 106, 219
FSC case study, 129–130, 144–145, 219–221
K-12 phenomenon, 36–39
mechanics. See Mechanics of communication
Mission Planning task, 210
organization capability, 97, 103
organization culture, 32
and overstaffing, 137
pair programming, 79
printed documentation, 115
product characteristics, 220–221
radiation, 24
software development teams, 87, 89–90, 99,
289
value, 283–284
utensils, 30
Compilers
defined, 315
directives for, 234
and structured programming, 60
Complex DISP (Special Display Requirements)
rating categories, 191
Complexity, 137
AEXP, 106
COCOMO definition, 140–141
code growth model, 243–246
defined, 315
first-order parametric models, 301–302
FSC case study, 143–145
quantitative complexity value, 138–142
in software equation, 164
Complexity (D)Rating Matrix, 142
Component-level estimates, 315
Computer data definitions, 315
Computer software, 315
Computer software component, 316
Computer Software Configuration Item (CSCI)
acceptance tests, 317
COTS impact, 231
defined, 316,
deleted source code, 231
modified source code, 230–231
new source code, 230
size specification, 56, 232–233
Computer Software Configuration Item (CSCI), continued
  source code elements, 228–229
  waterfall process, 207
Computer software units, 316
Communication tools, 29–30
Configuration items, 316
Confirmation bias
  as communication barrier, 285
  defined, 316
  impact, 34–36
  for structured programming, 34–35, 60
Constantine, Larry, 61
Constraints
  development, 16
  memory, 192–193
  TIMC (Timing Constraints), 195, 199
COnstructive COst MOdel. See COCOMO tools
Continuous design walkthroughs, 86, 289
Control centralized team structure, 74
Control in real-time operations, 194–195
Controlling organizations, 42, 285
Convection, information, 23–24
Cooperation in organization capability, 97
Coordination
  pair programming, 80
  software management for, 42
Corrective maintenance, 274
Cost
  as constraint, 16
  drivers, 11, 224, 282–283
  estimates. See Development schedule and cost estimates
  life-cycle, 273
  mastery, 105, 175
Cost Analysis Requirements Description (CARD), 231–232
Cost-estimating relationships (CERs), 298
COTS (Commercial Off-the-Shelf) products
  defined, 316
  size impact, 231
Countable SLOC, 234
Counting function points, 253, 259–260
Coupling, 316
Covert Agenda, 44
Critical Design Review (CDR), 208
Cross-functional teams, 81–82
  Basic technology constant, 122
defined, 316
Skunk Works, 82–83
study on, 83–84
CSCI. See Computer Software Configuration Item (CSCI)
CSCI acceptance tests, 317
Cube farms
  ad hoc teams, 73
  communication and collaboration in, 27–29, 51
  description, 19
Cubicle furniture system, 27–28
Culture issues, 31–32
  comfort zone, 33–34
  confirmation bias, 34–36
  K-12 phenomenon, 36–39
  modern practices, 107
  organization culture, 32–33
  shifts, 8
Cubicles as communication barriers, 14
Danger zone, 34
Dartmouth BASIC, 176–177
Data declarations, 234
Data dimensions in transitions, 264
Davis, Alan, 262
De facto standard tools, 112
Debugging programs
  DEBUG statements, 234–235
  third-generation languages, 59
Dedicated project areas, 29
Dekkers, C. A., 253
Deleted source code, 231
Delphi method, 296–297
DeMarco, Thomas
  Covert Agenda, 44
  investment in people, 31
  software dimensions, 255
  software system decomposition, 61
Deming, W. E., 43, 148
DePree, Hugh, 27
Design factor in effective size equation, 239
Designed SLOC, 233
DET field, 258–259
Detail design, 317
Developer capability
  evolving concept, 95–98
  stability in, 62
Development configuration, 317
Development constraints, 16
Development costs in software equation, 164–165
Development effort penalties, 106
Development environment, 173–174
Command Generation task, 216–217
Communication task, 219–220
defined, 317
experience factors, 174–182
management factors, 183–188
Mission Planning task, 210–212
volatility factors, 182–183
Development platforms, 317
Development Resource Dedication (RDED) rating, 187–188
Development schedule and cost estimates
Command Generation task, 215–218
Communications task, 219–221
development steps, 205–206
FSC case study, 208–210
Mission Planning system task, 210–215
productivity history for, 5
proposals, 15
Seer software model, 169–171
summary, 222–224
waterfall development model, 206–208
Development SLOC, 235
Development System Experience (DEXP) rating, 104, 175–176
Development System Volatility (DVOL) rating
Command Generation task, 216
Communication task, 219
Mission Planning task, 211–212
overview, 182–183
Development task problem in pair programming, 77
Development technology breakthroughs, 6–7
DEXP (Development System Experience) rating, 104, 175–176
Digital Equipment Corporation, 60–61
Dijkstra, Edsger, 60
Directing management style, 100
Directing organizations, 285
Discussion centers, 26
DISP (Special Display Requirements), 190–191, 220
Distance
as communication barrier, 25
information convection, 23
management, 33
MULT, 185
RLOC, 187
Theory X/Theory Y, 49
Distributed systems, 175, 180–181
Development Practices and Procedures for CMMI, 66
Documentation
heavyweight vs. lightweight development, 152
history of, 7
printed, 115–116
Documented SLOC, 233
as cost driver, 224
FSC case study, 131–132, 135
Mission Planning task, 210
organization capability, 103–107, 118–120
Doty Associates Model, 160–161, 300
Drafts, communication, 24–26
Drivers
cost, 11, 224, 282–283
defined, 317
pair programming, 13, 76, 78
DVOL (Development System Volatility) rating
Command Generation task, 216
Communication task, 219
Mission Planning task, 211–212
overview, 182–183
E-mail, 28
EAL (Evaluation Assurance Level) rating, 198
Early Design model, 206
ECSs (Embedded Computer Systems) defined, 318
function points, 258
Effective average experience, 105–106
AEXP, 106–107
DEXP, 175–176
LEXP, 176–179
PEXP, 179–180
TEXP, 180–182
Effective size, 317
Effective size estimation
effective size equation, 238–241
ESLOC, 235–238
FSC case study, 249–252
overview, 227–228
Effective size estimation, continued

- size growth, 241–247
- size risk, 248–249
- SLOC, 233–234
- in software equation, 168
- source code elements, 228–231
- uncertainty in, 231–233

Effective source lines of code (ESLOC), 228

- defined, 123, 317
- first-order parametric models, 301
- overview, 235–238

Effective source lines of code per person month (ESLOC/PM), 59

Effective technology constant

- in Basic technology constant, 200–201
- communication task, 220
- defined, 318
- Mission Planning task, 212–213
- in software equation, 168–169

Effectiveness Formula, xvi, 13, 280

- Application Domain Experience (AEXP) in, 103–107
- Analyst and Programmer Capability in, 99–103
- culture issues. See Culture issues defined, 318
- formula, 18–19

Hardcopy Turnaround Time (TURN), 114–116

- mechanics of communication. See Mechanics of communication
- modified Effectiveness Formula, 117
- organization capability (OCR), 94–95
- overview, 17–18, 98–99, 279–281
- and Seer, 162
- software engineer value, 71
- Terminal Response Time (RESP) rating, 114
- tiger teams, 85
- Use of Modern Practices (MODP) rating, 107–112
- Use of Modern Tools (TOOL) rating, 112–114
- Effectiveness of information transfer, 21
- Efficiency, 318
- EIFs (External Interface Files), 260
- Einstein, Albert, 173, 290–291
- Elementary processes in EIF files, 260
- Embedded computer systems (ECSs) defined, 318
- function points, 258
- Engineering graduates useful life expectancy, 8
- Enhancements, software, 275
- Entropy defined, 318
- second-order parametric models, 302–303

Environmental factors

- Basic technology constant, 123–124
- real-time operations, 195
- EO (External Output) with function points, 261
- Epimetheus, 58
- EQ (External Inquiry) with function points, 261–262
- Error rate in pair programming, 78
- ESLOC (effective source lines of code), 228
- defined, 317
- first-order parametric models, 301
- overview, 235–238

Estimation

- Communication task, 221
- cost and schedule. See Development schedule and cost estimates
- maintenance. See Maintenance estimating
- Mission Planning task, 212–214
- Seer. See Seer software model
- size. See Effective size estimation
- software. See Software estimating models
- tools, 5
- Eureka syndrome, 77
- Evaluation Assurance Level (EAL) rating, 198
- Events and responses in transitions, 264
- Evolutionary development, 318–319
- Executable statements, 234
- Exigence, 3
- Expansion ratio, 319
- Experience factors, 105–106, 174–175
- AEXP, 106–107
- Communication task, 219
- DEXP, 175–176
- LEXP, 176–179
- organization capability, 103–107
- pair programming, 78–81
- PEXP, 179–180
- TEXP, 180–182
- Expert judgment software estimating models, 295–297
- Experts, communication by, 106
- External environment in real-time operations, 195
External input with function points, 260–261
External inquiry (EQ) with function points, 261–262
External interface files (EIFs), 260
External output (EO) with function points, 261
Extreme Programming (XP)
Agile methods, 288
pair programming, 75

Familiar products, 196
Faultless Software Corporation (FSC) case study, 125
capability ratings, 130–136
Command Generation task, 215–218
Communication task, 219–221
complexity, 143–145
corporate background, 125–126
development schedule and cost estimates, 208–210
effective size estimation, 249–252
Hardcopy Turnaround Time (TURN) rating, 134
maintenance estimation, 277–278
Mission Planning system, 210–215
Use of Modern Practices (MODP) rating, 132–133
Use of Modern Tools (TOOL) rating, 133–134
problem, 125–126
software components, 128–130
Terminal Response Time (RESP) rating, 134
Fayol, Henry, 42
Federal Aviation Administration (FAA)-related software, 193
Feedback in communication, 22
Ferris, S. R., 21, 32
File Type Referenced (FTR), 260–261
Firmware, 319
First-order parametric models, 300–301
Flat staffing in Brooks’ Law, 155–156
Focused energy in teams, 86, 289
Foreign resources and support, 187
Formal Qualification Test (FQT), 149
FORMAT statements, 234
Four communication commandments, 24–31
Fourth-generation languages, 319
FPs. See Function point (FP) sizing
FQT (Formal Qualification Test), 149
FSC. See Faultless Software Corporation (FSC) case study
FTR (File Type Referenced), 260–261
Full-scale development, 319
Function Point Counting Practices Manual, 253
Function point (FP) sizing, 227–228, 253
Adjusted (AFP), 265–268
application boundaries, 256–257
backfiring, 268–270
component overview, 254–256
counting tips, 259–260
defined, 319
External Input (EI), 260–261
External Inquiries (EQ), 261–262
External Interface Files (EIF), 260
External Output (EO), 261
function point counting, 253
Internal Logical Files (ILF), 257–258
and objects, 270–271
rankings, 259
SLOC conversion, 269
Transforms (TF), 262–263
Transitions (TR), 263–264
Unadjusted Function Point (UFP) counting, 264–265
zero function point problem, 271–272
Furniture police, 28
Future Shock, 8
Gaffney, J. E., 227
General System Characteristics (GSCs)
defined, 266, 319
Value Adjustment Factor (VAF), 267–268
Gilb, T., 9
Gilb delivery method, 9
Glossary of terms, 313–326
GOTO statements, 34, 60
Graphical solutions in Seer model, 165–166
Grass-roots estimating, 297–298
Group dynamics, 43
Growth, 232
defined, 319
effective size estimation, 241–247
GSCs (General System Characteristics)
defined, 319
Value Adjustment Factor (VAF), 267–268
HAL 9000, 205
Hardcopy Turnaround Time (TURN)
FSC case study, 134–135
organizational capability, 114–120
Hatley, D. J., 61
Hatley/Popbabai method, 2, 61, 127
Hawthorne Effect
  defined, 319–320
  management importance, 98
  motivation in, 87, 285–286
  overview, 45–47
  study importance, 42–43
Heavyweight development staffing profiles, 151–152
Hephaestus, 58
Herman Miller Company, 27
Herzberg motivators, 43
Higher-order languages (HOL), 320
Historical size data, 241
History, learning from, 57
  CMMI, 63–66
  maximum/minimum schedule, 66–68
  Pandora and the Magic Vase, 58–59
  summary, 68–69
  technology productivity gains, 59–63
HOL (Higher-Order Languages), 320
Holchin, Barry, 242
Hope
  FSC case study, 133
  Pandora and the Magic Vase, 58
HOST (Rehosting Requirement), 191–192
Human behavior in Theory X/Theory Y, 47–49
Humphrey, Watts S.
  CMMI concept, 65–66, 102–103
  estimation techniques, 242
  process improvements, 148

Ideal intermolecular coupling, 235
IEEE 1207 standard, 194
IFPUG (International FP Users Group), 253
ILFs (Internal Logical Files), 257–258
Illumination in work area study, 45–47
Implementation factor in effective size equation, 239
INCLUDE statements, 234
Incremental software development
  Agile methods, 13–14
  defined, 320
Independent Verification and Validation (IV&V) requirements, 194
Information convection, 23–24
Information flow, 54–55
Information transfer, 284
Initial Operational Capability (IOC), 153
Initial penalty in programming languages, 178
Input/output (I/O) drivers, 320
Instructions, defined, 320
Interchangeable DISP category, 191
Intermolecular coupling, 235
Internal Logical Files (ILFs), 257–258
International FP Users Group (IFPUG), 253
Invisible walls, 25, 28
IOC (Initial Operational Capability), 153
IV&V (Independent Verification and Validation) requirements, 194
Jensen, Randall W.
  Effectiveness Formula, 71
  Seer development, 300
Jensen Software Development Model. See Seer model
Jet fighter planes prototypes, 83
Johnson, Clarence "Kelly" L., 83
Johnson, Kelly, 26–27, 83
Jones, Capers, 253, 268–269
K-12 phenomenon
  collaboration, 36–39
  defined, 320
  Knowledge retention, 276
  Known products, 196–197
Lack of experience in organization capability, 103–107
Languages, defined, 320
Leadership
  vs. management, 16
  teams, 87, 289
  Theory X/Theory Y. See Theory X and Theory Y
Leading management style, 100
Leading organizations, 285
Learning
  history. See History, learning from
  K-12 phenomenon, 36–39
Learning curves, 174
  organization capability, 104–106
  programming languages, 176–177
Levels of teams, 72
LEXP (Programming Language Experience) rating
Mission Planning system, *continued*
- estimates, 212–214
- FSC case study, 128–129, 143–144, 210–215
- product characteristics, 212–213
- size information, 210

Modern communications in organization culture, 32

Modern Practices (MODP) rating
- Command Generation task, 216
- Effectiveness Formula, 107–111
- FSC capability, 132–133, 135
- Mission Planning task, 211
- Organization Capability Rating, 118–120
- PEXP (Development Practices Experience) rating, 179

Modern tools and technology
- for communication barriers, 23
- Effectiveness Formula, 111–114
- FSC case study, 133–134

Modified code
- defined, 321
- SLOC, 230

Modified software systems in ESLOC, 236–237

MODP. See Modern Practices (MODP) rating

Modular design, 321

Molecular software view, 235–236

MORG (Multiple Development Organizations) rating
- description, 185
- Mission Planning task, 211–212

Motivation, 286–287
- Agile software development, 51
- in analyst and programmer capability, 100
- Hawthorne Effect. See Hawthorne Effect
- importance, 52–53
- software management for, 41
- teams, 87, 289–290
- Theory X/Theory Y, 47–49

Multiple Development Organizations (MORG) rating
- description, 185
- Mission Planning task, 211–212

Multiple Development Sites (MULT) rating
- Command Generation task, 217
- Communication task, 220
- description, 185–186
- Mission Planning task, 211–212

Multiple Security Classifications (MCLS) rating, 184–185

Multiuser systems in DEXP, 175, 180–181

Mythical Man-Month, *The*, 12–13

National Security Agency Orange Book, 198

NATO Software Engineering Conference, 2–3, 94

Navigators in pair programming, 13, 76, 78

NDS (nondevelopmental software), 321

Network communications, 28

New code
- defined, 321
- SLOC, 230

New technology, 197

Noise vs. collaboration, 25

Nondevelopmental software (NDS), 321

Norden, Peter V., 148, 160

Norden staffing profile, 299

Norden study, 148–151

Normal distributions, 232

Object-based languages, 7

Object programs, 321

Objects
- defined, 321
- and function points, 270–271

OCR (Organization Capability Rating), 116–120, 281

Online software development, 61

Operating systems
- defined, 321
- development of, 139–140

Operational software, 321

Optimal performance zones, 34

Optimistic utterings, 36

Orange Book ratings, 198

Organization capability measuring. See Measuring organization capability in software equation, 167

Organization Capability Rating (OCR), 116–120, 281

Organization culture, 32–33

Organizing organizations, 42, 285

Overlapping system development staffing profiles, 152–153

Overstaffing, 12, 74, 137, 139, 147, 154–158

Overtime, unpaid, 223
Pair programming
communications and management support, 10
defined, 321
development task problem, 77
experiment results, 78–79
experiment summary, 79–81
1975 experiment, 77–81
overview, 75–77
productivity contributions, 13
team composition, 77–78
Pandora and the Magic Vase, 58–59, 242
Paper in communication, 22
Parametric software estimating models, 294, 298–299
evolution, 299–300
first-order, 300–301
second-order, 302–304
third-order, 304–306
Parker, Glenn, 101
Parker's Law, 294
Part-time commitment, 31
Participative leadership, 87, 289
Pascal, Blaise, 17
Pascal language, 179
Patton, George S., Jr., 47
Paul Masson Point, 165–167, 170
Paul Masson Rule, 67
PCAP rating
and ACAP rating, 102
FSC case study, 135
Organization Capability Rating, 118–120
People
Agile software development, 50
productivity impacts, 9–13
sharing between projects, 31
in software equation, 164
People-Process-Project triad, 9–10, 44, 281–283
Perfective maintenance, 274
Perlis, Alan J., 159
Person month (PM), 322
Personal Software Process (PSP), 45
Personnel issues in Mission Planning task, 210
PEXP (practices and methods experience) rating
Command Generation task, 216
Communication task, 219
learning curve, 104
Mission Planning task, 211
overview, 179–180
Physical SLOC, 235
PI (productivity index) rating, 322
Pirbhai, I. A., 61
PL/I language, 177
Planning
FSC case study. See Mission Planning system organizations, 285
software management for, 42
PM (person month), 322
Popcorn, 30
Post-Architecture model (COCOMO II), 206
Post mortem analyses, 203
Practices and methods experience (PEXP) rating
Command Generation task, 216
Communication task, 219
learning curve, 104
Mission Planning task, 211
overview, 179–180
Practices and Methods Volatility (PVOL) rating
Command Generation task, 216
Communication task, 219
overview, 182–183
Predictive approach in heavyweight development, 152
Preliminary Design Review (PDR), 208
Price-S tool, 5
Price-to-win software development, 294
Prima donna programmers, 80
Printed documentation, 115–116
Problem isolation in teams, 87, 289
Problem-solving skills in team environments, 99
Process maturity, 63
Process terminal in People-Process-Project triad, 9
Processing steps transforms, 262
Product characteristics, 189–190
Basic technology constant projection, 201–203
Command Generation task, 218
Communication task, 220–221
Effective technology constant, 200–201
memory constraint, 192–193
Mission Planning task, 212–213
quality assurance requirements, 193–194
Product characteristics, continued

Real-Time Operations (RTIM) rating, 194–195
Rehosting Requirement (HOST) rating, 191–192
Required Testing Level (TEST) rating, 199
Requirements Volatility (RVOL) rating, 195–197
Software Security Requirements (SECR) rating, 197–198
Special Display Requirements (DISP) rating, 190–191
System CPU Timing Constraint (TIMC), 199
System Specification Level (SPEC) rating, 198–199
Target System Volatility (TVOL) rating, 199–200
Productivity index (PI), 322
Productivity overview, 1–9, 51
Agile contributions, 13–14
Basic technology constant, 123
CMMI, 63–66
constraints, 16
defined, xv, 322
first-order parametric models, 301
Hawthorne Effect, 45–47
historic growth, 61–63
information flow, 54–55
magic bullets, 15
Organization Capability Rating (OCR), 117
people impact, 9–13
and software estimating tools, 4–5
technology impact, 52–54, 59–63
Programmer Capability (PCAP) rating
FSC case study, 130–131
overview, 99–103
Programmer's Workbench (PWB), 2, 60–61
Programming Language Experience (LEXP) rating
learning curve, 104
Mission Planning task, 211
overview, 176–179
Programs, defined, 322
Project areas, dedicated, 29
Project terminal in People-Process-Project triad, 9
Prometheus, 58
Propst, Robert, 27–28
PSP (Personal Software Process), 45
Putnam, Larry H.
complexity data, 140–141
cost estimating model, 138
and Seer model, 160–162
SLIM model, 299
PVOL (Practices and Methods Volatility) rating
Command Generation task, 216
Communication task, 219
overview, 182–183
PWB (Programmer's Workbench), 2, 60–61
Quality as constraint, 16
Quality Assurance Requirements (QUAL) rating
Command Generation task, 218
Communication task, 221
Mission Planning task, 213
overview, 193–194
Quantitative complexity value, 138–142
Quiet environments for development teams, 88
Rabble hypothesis, 43
Radiation, 24
Ranks
function points, 259
transforms, 262
Rate of technology change, 67
Rayleigh development model, 152–154
Rayleigh staffing profile
defined, 322–323
description, 149–150
RDED (Resource Dedication) rating, 187–188
Real time, defined, 322–323
Real-Time Operations (RTIM) rating, 194–195
Real-time systems
airborne weapons, 7
function points, 258
Regression tests
effective size estimation, 237–239
source code, 231, 247
Rehosting Requirement (HOST) rating, 191–192
Required Testing Level (TEST) rating
Command Generation task, 218
Communication task, 221
description, 199
Mission Planning task, 213
Requirements analysis, 323
Requirements change, project failures from, 196
Requirements Volatility (RVOL) rating
Command Generation task, 218
Communication task, 221
Mission Planning task, 213
overview, 195–197

Resource and Support Location (RLOC) rating
Command Generation task, 216
Communication task, 219
Mission Planning task, 211
overview, 186–187

Resource Dedication (RDED) rating, 187–188

Resource sharing, 31

RESP (terminal response time) rating, 114
FSC case study, 134–135
Organization Capability Rating, 118–120

RET (record element type), 258–259

Reused source code
defined, 323
description, 231
in effective size equation, 240–241

Risk, size, 248–249

RLOC (Resource and Support Location) rating
Command Generation task, 216
Communication task, 219
Mission Planning task, 211
overview, 186–187

RTIM (Real-Time Operations) rating, 194–195

RVOL (Requirements Volatility) rating
Command Generation task, 218
Communication task, 221
Mission Planning task, 213
overview, 195–197

SAF (Size Adjustment Factor)
defined, 323
Seer model, 238

Sage model. See also Seer model
Basic technology constant, 123
Capability Calculator based on, 119
Complexity value, 243
development of, 160
effective size equation, 239
evolution of, 300
usefulness, 5

Schedules
and complexity, 140
as constraint, 16
and cost estimates. See Development schedule and cost estimates
estimation proposals, 15
productivity history for, 5
Seer software model, 169–171
in software equation, 165
Scope as constraint, 16
SDD (software design document), 324
Seating arrangements, 25
Second Law of Consulting, 10, 44
Second-order parametric models, 302–304
SECR (Software Security Requirements) rating, 197–198
Security, defined, 323
Seer model, 159
Basic technology constant, 121, 123
development of, 300
entropy values, 303
environmental factors, 304
introduction, 160–162
organization capability, 93, 97–98
SAF in, 238
schedule and cost estimates, 169–171
software equation, 163–169
usefulness, 5

SEER-SEM model
Basic technology constant, 123
Capability Calculator based on, 119
code growth, 243
commercial implementation, 300
Complexity value, 243
effective size equation, 239
usefulness, 5

Semantic statements, 262

Separation distance
as communication barrier, 25
information convection, 23
management, 33
MULT rating, 185, 212, 217, 220
RLOC rating, 187, 211, 216
Theory X/Theory Y, 49

Sharing resources, 31
Sheepherders, 53
Shepherds, 53
Silver bullets in software equation, 167
Simple DISP category, 190
Single-user systems in DEXP, 175–176, 180–181
Six Sigma, 148

Size
Command Generation task, 215
Size, continued
Communication task, 219
effective. See Effective size estimation
function point. See Function point (FP) sizing
Mission Planning task, 210
single development tasks, 67
software development teams, 88–91
in software equation, 167–168
Size Adjustment Factor (SAF)
defined, 323
Seer model, 238
Size risk, 248–249
Skinner, B. F., 189
Skunk Works
Basic technology constant, 122
defined, 323–324
description, 26–27, 82–83
SLIM (Software Lifecycle Model), 243, 299
SLOC. See Source lines of code (SLOC)
Software crisis
defined, 324
overview, 2–9
Software design document (SDD), 324
Software development issues, 1–2
  Agile productivity contributions, 13–14
development constraints, 16
  magic bullets, 15
  people impact on productivity, 9–13
  software crisis, 2–9
Software development model, 16
Software development plans, defined, 324
Software development teams, 71–72
  ad hoc, 73–74
  chief programmer, 74–75
cross-functional, 81–84
definitions, 72
environment effects, 88–91
K-12 phenomenon, 38–39
negative effects, 87–88
overview, 72–73
pair programming, 75–81
positive effects, 86–87, 288–290
size, 88–91
Skunk Works, 82–83
summary, 287–290
team utensils, 30, 82
tiger, 84–85
tiger, definition, 84
tiger, Agile development, 84
virtual, 14
Software documentation, defined, 324
Software Effectiveness Formula. See
  Effectiveness Formula
Software engineering environment, 324
Software Engineering Institute research, 11
Software engineering term, 3
Software enhancement, 275
Software equation
  Basic technology constant projection,
    201–202
effective size, 168
  Effective technology constant, 168–169
  overview, 163–168
Software estimating models
  analogy, 294–295
  bottom-up, 297–298
  expert judgment, 295–297
  overview, 293–294
  parametric, 298–306
Software Lifecycle Model (SLIM), 243, 299
Software maintenance, defined, 324–325
Software management. See Management
Software process maturity levels, 64
Software quality, defined, 325
Software Security Requirements (SECR) rating,
  197–198
Software spiral method, 8–9
Source code elements, 228–231
Source lines of code (SLOC), 229
defined, 227, 325
first-order parametric models, 300
function point conversion to, 268–270
modified source code, 230
overview, 233–234
as standard measure of product size, 5
total SLOC, 231
Spaghetti code, 235
Spanish theory of management, 223
SPEC (System Specification Level) rating
  Command Generation task, 218
  Communication task, 221
  Mission Planning task, 213
  overview, 198–199
Special Display Requirements (DISP) rating,
  190–191
Staffing and staffing profiles, 147
effective use of people, 147–151
lightweight vs. heavyweight development,
  151–152
management function, 285
overstaffing, 12, 74, 137, 139, 147, 154–158
Rayleigh development model, 152–154
in software equation, 164
Standish CHAOS Manifesto, 3
State-transition diagrams, 264
Steady-state maintenance effort, 277
Stratification of system complexity values, 139
Structured Analysis and System Specification, 61
Structured Design, 61
Structured programming
confirmation bias, 34
productivity improvements, 60
Support software, 325
System CPU Timing Constraint (TIMC) rating, 199
System Evaluation and Estimation of Resources
Software Estimating Model. See SEER-SEM model
System software, defined, 325
System Specification Level (SPEC)
Command Generation task, 218
Communication task, 221
Mission Planning task, 213
overview, 198–199
Target System Experience (TEXP)
learning curve, 104
overview, 180–182
Target System Volatility (TVOL), 199–200
Task complexity in AEXP, 106
Team Software Process (TSP), 45
Teams. See Software development teams
Technical talent in Effectiveness Formula, 18
Technology
for creative work, 30
productivity improvements, 52–54, 59–63
unfamiliar, 197
Terminal Response Time (RESP) rating, 114
FSC case study, 134–135
Organization Capability Rating (OCR), 118–120
Terminology, 313–326
TEST (Required Testing Level) rating
Command Generation task, 218
communication task, 221
description, 199
Mission Planning task, 213
Test factor in effective size equation, 239–240
Tested SLOC, 233
TEXP (Target System Experience) rating
learning curve, 104
overview, 180–182
Theory X and Theory Y
defined, 286–287, 325
development of, 42–43
motivation in, 52–53
organization capability, 97, 100
organization culture, 32
overview, 47–49
summary, 286–287
Third-generation programming languages, 7, 59
Third Law, 15
Third-order parametric models, 304–306
Tiger teams
ad hoc, 73–74
and chaos, 148
description, 84–85
Tight coupling, 235
Time constraint parameter (TIMC) rating, 195, 199
Time for problem resolution, 54
Time frame for development tasks, 67
Toffler, Alvin, 7–8
Tonies, Chuck, 17, 71
TOOL (Use of Modern Tools) rating, 112
FSC case study, 133, 135
Organization Capability Rating, 118–120
Tools
Agile software development, 50
for creative work, 29–30
Effectiveness Formula, 111–114
FSC case study, 133–134
Top-down software development, 294
Total SLOC, 231
Toynbee, Arnold, 57
Training for programming languages, 177
Transactions
defined, 325
function points, 254
Transforms (TFs) for function points, 255, 262–263
Transitions (TRs) for function points, 255, 263–264
TSP (Team Software Process), 45
TURN (Hardcopy Turnaround Time) rating
FSC case study, 134–135
Organizational Capability rating, 114–120
TVOL (Target System Volatility) rating, 199–200
Two-person programming. See Pair programming

UML (Universal Modeling Language), 7, 63
Unadjusted function points (UFPs) counting, 264–265
defined, 325
Uncertainty in effective size estimation, 231–233
Unfamiliar technology, 197
United States Army Computer Systems Command (USACSC) database, 161–162
Universal Modeling Language (UML), 7, 63
Unpaid overtime, 223
Updating CMMI, 66
USACSC (United States Army Computer Systems Command) database, 161–162
Useful life expectancy of college engineering graduates, 8
User-friendly DISP category, 190
Utensils
creative work, 29–30
cross-functional teams, 82

Value Adjustment Factor, 266–268, 325
Virtual teams, 14
Virtual walls, 25, 28
Visually radial communication, 24
Vocal tonality in communication, 21
Vocally radial communication, 24
Volatility
code growth, 241
Command Generation task, 216
in development environment, 182–183, 219
Mission Planning task, 211–213, 224
of requirements, 195–197
in software equation, 167
target system, 199–200
von Neumann, John, 178
Von Neumann thinking, 178–179

Walls, invisible, 25, 28
War rooms in pair programming, 80
Ward, P. T., 61
Waterfall-based projects
basic technology constant, 121
defined, 326
as early standard, 8
overview, 206–208
People-Process-Project triad, 9
WBS (work breakdown structure), 298
Weapons system, 7
Weinberg, Gerry M.
cost drivers, 11, 282–283
hope for improvements, 273
management importance, 12
people problems, 98
programmer intelligence, 37
Second Law of Consulting, 10, 44
Western Electric Company facility, 42–43
White box elements, 229–230
Whitmire, Steve, 253
Wideband Delphi method, 297
Wilkes, Maurice, 8, 37
Wishful thinking
in confirmation bias, 35
in software equation, 167
Wolverton, Ray, 160, 299–300
Words, defined, 326
Work breakdown structure (WBS), 298
Working smarter, 223
Workspace for cross-functional teams, 82
Workstations, 61

XP (extreme programming)
Agile methods, 288
pair programming, 75
XP-80 Shooting Star jet fighter, 83

Yourdon, Ed, 61

Zero function point problem, 271–272
Zeus, 58