IMPROVING SOFTWARE DEVELOPMENT PRODUCTIVITY
Effective Leadership and Quantitative Methods in Software Management

Randall W. Jensen
Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the United States, please contact international@pearsoned.com.

Visit us on the Web: informit.com/ph

Library of Congress Cataloging-in-Publication Data

Jensen, Randall W.
Improving software development productivity : effective leadership and quantitative methods in software management / Randall W. Jensen, Ph.D.
    pages cm
    Includes index.
    1. Computer software—Development. I. Title.
    QA76.76.D47J46 2015
    005.1—dc23
    2014019684

Copyright © 2015 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.

ISBN-10: 0-13-356267-0

Text printed in the United States on recycled paper at Courier in Westford, Massachusetts.

First printing, August 2014
# 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
Chapter 8: Product Complexity
  8.1 A Quantitative Complexity Value
  8.2 Faultless Software Complexity Example
    8.2.1 Mission Planning System
    8.2.2 Command Generation System
    8.2.3 Communication System

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

Chapter 10: Seer Software Model Introduction
  10.1 Introduction
  10.2 Software Equation
    10.2.1 Effective Size
    10.2.2 Effective Technology Constant
  10.3 Schedule and Cost Estimates

Chapter 11: Development Environment
  11.1 Experience Factors
    11.1.1 Development System Experience (DEXP)
    11.1.2 Programming Language Experience (LEXP)
    11.1.3 Practices and Methods Experience (PEXP)
    11.1.4 Target System Experience (TEXP)
  11.2 Volatility Factors
    11.2.1 Development System Volatility (DVOL)
    11.2.2 Practices and Methods Volatility (PVOL)
11.3 Management Factors ........................................... 183
  11.3.1 Multiple Security Classifications (MCLS) ............ 184
  11.3.2 Multiple Development Organizations (MORG) .... 184
  11.3.3 Multiple Development Sites (MULT) ................. 185
  11.3.4 Resource and Support Location (RLOC) .............. 186
  11.3.5 Development Resource Dedication (RDED) .......... 187

Chapter 12: Product Characteristics .............................. 189
  12.1 Product Characteristics Evaluation ....................... 190
    12.1.1 Special Display Requirements (DISP) ............ 190
    12.1.2 Rehosting Requirements (HOST) ................. 191
    12.1.3 Memory Constraints (MEMC) .................... 192
    12.1.4 Quality Assurance Requirements (QUAL) ......... 193
    12.1.5 Real-Time Operations (RTIM) .................... 194
    12.1.6 Requirements Volatility (RVOL) ................ 195
    12.1.7 Software Security Requirements (SECR) .......... 197
    12.1.8 System Specification Level (SPEC) .............. 198
    12.1.9 Required Testing Level (TEST) ................. 199
    12.1.10 System CPU Timing Constraints (TIMC) ........ 199
    12.1.11 Target System Volatility (TVOL) ............. 199
  12.2 Effective Technology Constant .......................... 200
  12.3 Basic Technology Constant Projection .................. 201

Chapter 13: Development Schedule and Cost Estimates ........... 205
  13.1 Waterfall Development Model ............................. 206
  13.2 Faultless Software Case Study .......................... 208
    13.2.1 Mission Planning ............................... 210
    13.2.2 Command Generation .............................. 215
    13.2.3 Communication .................................. 219
    13.2.4 Looking at the Big Picture ..................... 222
  13.3 Summary .................................................... 225

Chapter 14: Effective Size Estimation ............................ 227
  14.1 Source Code Elements .................................... 228
    14.1.1 Black Box versus White Box Elements ........... 229
    14.1.2 New Source Code .................................. 230
    14.1.3 Modified Source Code ............................ 230
    14.1.4 Deleted Source Code ............................. 231
    14.1.5 Reused Source Code .............................. 231
    14.1.6 COTS Software ................................. 231
    14.1.7 Total SLOC .................................... 231
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>14.2</td>
<td>Size Uncertainty</td>
<td>231</td>
</tr>
<tr>
<td>14.3</td>
<td>Source Line of Code (SLOC)</td>
<td>233</td>
</tr>
<tr>
<td></td>
<td>14.3.1 Executables</td>
<td>234</td>
</tr>
<tr>
<td></td>
<td>14.3.2 Data Declarations</td>
<td>234</td>
</tr>
<tr>
<td></td>
<td>14.3.3 Compiler Directives</td>
<td>234</td>
</tr>
<tr>
<td></td>
<td>14.3.4 Format Statements</td>
<td>234</td>
</tr>
<tr>
<td>14.4</td>
<td>Effective Source Lines of Code (ESLOC)</td>
<td>235</td>
</tr>
<tr>
<td>14.5</td>
<td>Effective Size Equation</td>
<td>238</td>
</tr>
<tr>
<td></td>
<td>14.5.1 Design Factor</td>
<td>239</td>
</tr>
<tr>
<td></td>
<td>14.5.2 Implementation Factor</td>
<td>239</td>
</tr>
<tr>
<td></td>
<td>14.5.3 Test Factor</td>
<td>239</td>
</tr>
<tr>
<td>14.6</td>
<td>Size Growth</td>
<td>241</td>
</tr>
<tr>
<td>14.7</td>
<td>Size Risk</td>
<td>248</td>
</tr>
<tr>
<td>14.8</td>
<td>Faultless Software Size Example</td>
<td>249</td>
</tr>
<tr>
<td></td>
<td>14.8.1 Mission Planning System</td>
<td>249</td>
</tr>
<tr>
<td></td>
<td>14.8.2 Command Generation</td>
<td>250</td>
</tr>
<tr>
<td></td>
<td>14.8.3 Communication</td>
<td>251</td>
</tr>
<tr>
<td>Chapter 15: Function Point Sizing</td>
<td>253</td>
<td></td>
</tr>
<tr>
<td>15.1</td>
<td>Function Point Counting</td>
<td>254</td>
</tr>
<tr>
<td>15.2</td>
<td>Function Point Components</td>
<td>254</td>
</tr>
<tr>
<td></td>
<td>15.2.1 Application Boundary</td>
<td>256</td>
</tr>
<tr>
<td></td>
<td>15.2.2 Internal Logical File</td>
<td>257</td>
</tr>
<tr>
<td></td>
<td>15.2.3 External Interface File</td>
<td>260</td>
</tr>
<tr>
<td></td>
<td>15.2.4 External Input</td>
<td>260</td>
</tr>
<tr>
<td></td>
<td>15.2.5 External Output</td>
<td>261</td>
</tr>
<tr>
<td></td>
<td>15.2.6 External Inquiry</td>
<td>261</td>
</tr>
<tr>
<td></td>
<td>15.2.7 Transforms</td>
<td>262</td>
</tr>
<tr>
<td></td>
<td>15.2.8 Transitions</td>
<td>263</td>
</tr>
<tr>
<td>15.3</td>
<td>Unadjusted Function Point Counting</td>
<td>264</td>
</tr>
<tr>
<td>15.4</td>
<td>Adjusted Function Points</td>
<td>265</td>
</tr>
<tr>
<td></td>
<td>15.4.1 Value Adjustment Factor</td>
<td>266</td>
</tr>
<tr>
<td></td>
<td>15.4.2 Adjusted Function Point Calculation</td>
<td>268</td>
</tr>
<tr>
<td>15.5</td>
<td>Backfiring</td>
<td>268</td>
</tr>
<tr>
<td>15.6</td>
<td>Function Points and Objects</td>
<td>270</td>
</tr>
<tr>
<td>15.7</td>
<td>Zero Function Point Problem</td>
<td>271</td>
</tr>
<tr>
<td>Chapter 16: Maintenance Estimating</td>
<td>273</td>
<td></td>
</tr>
<tr>
<td>16.1</td>
<td>Introduction</td>
<td>273</td>
</tr>
<tr>
<td>16.2</td>
<td>Software Enhancement</td>
<td>275</td>
</tr>
</tbody>
</table>
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

\[
V = C/M(T)
\]

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 non-software 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
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.”

LEDWIS 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. At this conference, a list of software problems was presented as the major development concerns. The problem list included software that was

---

• 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 exigence² 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,³ 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

Figure 1.1  CHAOS 2012 Software project survey results

Software Crisis

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)
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 is 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
Future Shock\textsuperscript{10} as a form of time compression that is present as a rapid acceleration of requirements changes. Technology changes fed on themselves to produce an ever-increasing frequency of change. Agile software development represents a major developmental cultural change.

The culture shift is not only happening in the development culture, but also in the people (programmers, designers, etc.) who make up the development culture. At the time Future Shock was published, college engineering graduates had a useful life expectancy of about five years before their skills would be outdated; that is, unless these engineers continued their schooling. That useful life is now less than two years in the software world. New college graduates live in an environment where all problems should be solvable in an hour or two. New tools and languages make it possible to create web pages and phone apps almost overnight. Software development is drifting toward artistic design rather than software engineering. This shift is creating a cultural problem for the large-scale system developers.

The work required to produce a line of source code remains almost independent of the computing power packed in one source line. Producing the instructions with the keyboard is not where the work happens. The work is in interpreting the requirements, producing and testing the design, correcting faulty reasoning, coordinating the work with others working on the task, and producing the documentation. An important related comment that has been attributed to Maurice Wilkes, the 1967 Turing Award winner, is:\textsuperscript{11}

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


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.

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.

Gerry Weinberg derived a comparison of the relative impact of the development environment to cost and productivity from Barry Boehm’s classic study of software development economics.

---

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 1.5  Relative impact of four categories of software cost drivers according to the Boehm software engineering economics study

Source: Reprinted by permission of Dorset House Publishing from Gerald M. Weinberg, Quality Software Management, Vol. 3: Congruent Action, Fig. 2-1, p. 14. Copyright © 1994. All rights reserved.

Figure 1.6  Number of publications compared with Weinberg’s relative productivity impact

Source: Reprinted by permission of Dorset House Publishing from Gerald M. Weinberg, Quality Software Management, Vol. 3: Congruent Action, Fig. 2-2, p. 15. Copyright © 1994. All rights reserved.

software engineering economics.\(^{24}\) 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\(^ {25}\) 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:26

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.”27


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 Cockburn28 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.
Chapter 2

The Effectiveness Formula

There are two dangerous extremes.
One is to shut reason out,
the other is to let nothing in.
Blaise Pascal, Pensées

Thirty-five years ago Chuck Tonies and I wrote about the current state of software engineering as a basis for what Chuck liked to call the “Effectiveness Formula.”¹ As I wrote this text about improving software productivity and reviewed our introduction to Software Engineering, I was struck by the similarity between our description of software engineering in 1975 and the state of the profession today. I am extracting the material that introduces the Effectiveness Formula in its entirety from Software Engineering because of its important effect on software development productivity. A clearer, more practical description of software engineering would be hard to find.

There are a few software-related positions in industry that do not require frequent interaction with other people. They are rare. Pure research assignments and one-person development tasks represent a very small fraction of the total activity within the data-processing sector of the industry. The typical software engineering position is highly interactive, because the typical software development environment is highly interactive. The user, the customer, the project management, the analyst/designers, the programmers, the test engineers, and others are all involved in a loosely knit team during one or more phases of every project life cycle. Software engineers may appear in any of these roles, but no matter which assignment they have, they will interact with all the other members of

the team. Literally every day the engineers will find it necessary to stay aware of the activities of those around them and to understand the significance of each of those activities. They will find it necessary to understand and to act in concert with the project management plan, communicating coherently with a variety of individuals.

If software engineers are not capable of participating or are not motivated to participate in the inevitable ebb and flow of management decisions, and if they are not capable of communicating daily with members of the team, their contributions (no matter how brilliant) will be diminished, because they will, in all probability, not match the real product requirements.

### 2.1 The Formula

Software system development is a dynamic activity. No matter how effective our baseline and configuration control methods and no matter how stable the project staff, some degree of rethinking, replanning, redefining, and redirection are necessary as the project proceeds. Furthermore, communications among the team members are not always perfect, just as the best football teams lose yardage due to missed assignments. The incomplete and incorrect understanding of requirements, designs, and specific interfaces is inevitable. In fact, it is quite common. Frequent communications among all participants on a software development project is the only way that misunderstandings can be corrected. The process of achieving coherent communications along all the required paths is an iterative one.

For those reasons the software engineer’s value to an organization operating in the industrial environment is dependent on three attributes. These attributes are technical talent, the ability to understand management concepts, and the ability to communicate. All three are so intimately involved in the software engineering process that the net effect of an individual’s effort is best represented by his or her product:

$$E = C[M(CS)]$$

where

- $E = \text{net effectiveness (0-1)}$
- $C = \text{communication ability and skills (0-1)}$
- $M = \text{management concept awareness (0-1)}$
- $CS = \text{computer science technical ability (0-1)}$
Notice in the formula that \( M \) operates on \( CS \) (Technology), and \( C \) operates on \( M \). The intent of this notation is to emphasize the chaining effect of the calculation. The result is still zero if any of the terms are zero, but communications is not directly related to technology.

The relationship in Equation (2.1) defines the Effectiveness Formula. Our experience in the software industry, and especially in product-oriented environments, has shown the Effectiveness Formula to be a realistic model of software engineering performance. While it is true that we are still in an age of technical specialization, it is also true that software development work is by its very nature a complex interactive process. It requires careful, intense management, and even the most specialized of the contributors must act in concert with his or her colleagues and the management plan if the development process is to be efficient. The effectiveness formula in Equation (2.1) shows that if any of the three elements are missing the effectiveness approaches zero.

What I have just described is not the norm for current traditional software development physical environments. These environments have not been designed, by intent or by chance, to foster interactive communications among the project participants. Their purpose appears to even the casual observer as a means to prevent communications. The norm for modern large-scale software development environments is what I will refer to throughout the text as a “cube farm,” in which all communication is forced through the organization’s network. Communication, which has a range of values between zero and, ideally, unity, has an effective value of only 0.07 in the cube farm. With a value that low, a management effectiveness value of 1.0 and a perfect technology (1.0) cannot provide much of a contribution to the engineer’s effectiveness.

Our first discussion topic must be the important issue of communication effectiveness.

2.2 Mechanics of Communication

The software development industry has pursued many technology approaches for improved productivity over the past 40 years. Communication and collaboration issues cannot be resolved with the next silver bullet (technology tool). Yet, in the Effectiveness Formula, communication skills ranks first, and most important, of the three effectiveness factors.

Recognizing the importance of good management in software development productivity is only the first step in process improvement. Moreover, good management includes more than management style and organizational ability. Good management requires effective communication. Effective communication is thus essential to successful software development productivity gains.