Trustworthy Systems
Through Quantitative
Software Engineering

Lawrence Bernstein
C. M. Yuhas
Trustworthy Systems Through Quantitative Software Engineering
IEEE Computer Society Publications

The world-renowned IEEE Computer Society publishes, promotes, and distributes a wide variety of authoritative computer science and engineering texts. These books are available from most retail outlets. Visit the CS Store at http://computer.org/cspress for a list of products.

IEEE Computer Society/Wiley Partnership

The IEEE Computer Society and Wiley partnership allows the CS Press authored book program to produce a number of exciting new titles in areas of computer science and engineering with a special focus on software engineering. IEEE Computer Society members continue to receive a 15% discount on these titles purchased through Wiley or at wiley.com/ieeecs.

To submit questions about the program or send proposals please e-mail dplummer@computer.org or write to Books, IEEE Computer Society, 100662 Los Vaqueros Circle, Los Alamitos, CA 90720-1314. Telephone +1-714-821-8380.

Additional information regarding the Computer Society authored book program can also be accessed from our web site at http://computer.org/cspress.
Trustworthy Systems
Through Quantitative
Software Engineering

Lawrence Bernstein
C. M. Yuhas

A John Wiley & Sons, Inc., Publication
To our sons, daughters-in-law, and grandson
Contents

PREFACE xvii
ACKNOWLEDGMENT xxv

PART 1 GETTING STARTED 1

1. Think Like an Engineer—Especially for Software 3
   1.1 Making a Judgment / 4
   1.2 The Software Engineer’s Responsibilities / 6
   1.3 Ethics / 6
   1.4 Software Development Processes / 11
   1.5 Choosing a Process / 12
      1.5.1 No-Method “Code and Fix” Approach / 15
      1.5.2 Waterfall Model / 16
      1.5.3 Planned Incremental Development Process / 18
      1.5.4 Spiral Model: Planned Risk Assessment-Driven Process / 18
      1.5.5 Development Plan Approach / 23
      1.5.6 Agile Process: an Apparent Oxymoron / 25
   1.6 Reemergence of Model-Based Software Development / 26
   1.7 Process Evolution / 27
   1.8 Organization Structure / 29
   1.9 Principles of Sound Organizations / 31
   1.10 Short Projects—4 to 6 Weeks / 33
1.10.1 Project 1: Automating Library Overdue Book Notices / 33
1.10.2 Project 2: Ajax Transporters, Inc. Maintenance Project / 34

1.11 Problems / 35

2. People, Product, Process, Project—The Big Four / 39

2.1 People: Cultivate the Guru and Support the Majority / 40
2.1.1 How to Recognize a Guru / 41
2.1.2 How to Attract a Guru to Your Project / 42
2.1.3 How to Keep Your Gurus Working / 43
2.1.4 How to Support the Majority / 43

2.2 Product: “Buy Me!” / 45
2.2.1 Reliable Software Products / 46
2.2.2 Useful Software Products / 47
2.2.3 Good User Experience / 48

2.3 Process: “OK, How Will We Build This?” / 49
2.3.1 Agile Processes / 49
2.3.2 Object-Oriented Opportunities / 53
2.3.3 Meaningful Metrics / 60

2.4 Project: Making It Work / 61
2.5 Problems / 65
2.6 Additional Problems Based on Case Studies / 67

PART 2 ETHICS AND PROFESSIONALISM / 73

3. Software Requirements / 75

3.1 What Can Go Wrong With Requirements / 75
3.2 The Formal Processes / 76
3.3 Robust Requirements / 81
3.4 Requirements Synthesis / 84
3.5 Requirements Specification / 86
3.6 Quantitative Software Engineering Gates / 87
3.7 sQFD / 88
3.8 ICED-T Metrics / 91
3.8.1 ICED-T Insights / 92
3.8.2 Using the ICED-T Model / 94
3.9 Development Sizing and Scheduling With Function Points / 95
3.9.1 Function Point Analysis Experience / 95
3.9.2 NCSLOC vs Function Points / 96
3.9.3 Computing Simplified Function Points (sFP) / 97
3.10 Case Study: The Case of the Emergency No-Show Service / 98
3.11 Problems / 103

4. Prototyping 107

4.1 Make It Work; Then Make It Work Right / 107
   4.1.1 How to Get at the Governing Requirements / 108
   4.1.2 Rapid Application Prototype / 108
   4.1.3 What’s Soft Is Hard / 110
4.2 So What Happens Monday Morning? / 111
   4.2.1 What Needs to Be Prototyped? / 111
   4.2.2 How Do You Build a Prototype? / 112
   4.2.3 How Is the Prototype Used? / 112
   4.2.4 What Happens to the Prototype? / 114
4.3 It Works, But Will It Continue to Work? / 116
4.4 Case Study: The Case of the Driven Development / 116
   4.4.1 Significant Results / 119
   4.4.2 Lessons Learned / 122
   4.4.3 Additional Business Histories / 123
4.5 Why Is Prototyping So Important? / 128
4.6 Prototyping Deficiencies / 130
4.7 Iterative Prototyping / 130
4.8 Case Study: The Case of the Famished Fish / 131
4.9 Problems / 133

5. Architecture 137

5.1 Architecture Is a System’s DNA / 137
5.2 Pity the Poor System Administrator / 139
5.3 Software Architecture Experience / 141
5.4 Process and Model / 142
5.5 Components / 144
   5.5.1 Components as COTS / 144
   5.5.2 Encapsulation and Abstraction / 145
   5.5.3 Ready or Not, Objects Are Here / 146
5.6 UNIX / 148
5.7 TL1 / 149
   5.7.1 Mission / 150
   5.7.2 Comparative Analysis / 151
   5.7.3 Message Formatting / 152
   5.7.4 TL1 Message Formulation / 152
   5.7.5 Industry Support of TL1 / 152
5.8 Documenting the Architecture / 153
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.8.1</td>
<td>Debriefing Report</td>
<td>154</td>
</tr>
<tr>
<td>5.8.2</td>
<td>Lessons Learned</td>
<td>154</td>
</tr>
<tr>
<td>5.8.3</td>
<td>Users of Architecture Documentation</td>
<td>154</td>
</tr>
<tr>
<td>5.9</td>
<td>Architecture Reviews</td>
<td>155</td>
</tr>
<tr>
<td>5.10</td>
<td>Middleware</td>
<td>156</td>
</tr>
<tr>
<td>5.11</td>
<td>How Many Times Before We Learn?</td>
<td>158</td>
</tr>
<tr>
<td>5.11.1</td>
<td>Comair Cancels 1100 Flights on Christmas 2004</td>
<td>158</td>
</tr>
<tr>
<td>5.11.2</td>
<td>Air Traffic Shutdown in September 2004</td>
<td>159</td>
</tr>
<tr>
<td>5.11.3</td>
<td>NASA Crashes into Mars, 2004</td>
<td>159</td>
</tr>
<tr>
<td>5.11.4</td>
<td>Case Study: <em>The Case of the Preempted Priorities</em></td>
<td>160</td>
</tr>
<tr>
<td>5.12</td>
<td>Financial Systems Architecture</td>
<td>163</td>
</tr>
<tr>
<td>5.12.1</td>
<td>Typical Business Processes</td>
<td>163</td>
</tr>
<tr>
<td>5.12.2</td>
<td>Product-Related Layer in the Architecture</td>
<td>164</td>
</tr>
<tr>
<td>5.12.3</td>
<td>Finding Simple Components</td>
<td>165</td>
</tr>
<tr>
<td>5.13</td>
<td>Design and Architectural Process</td>
<td>166</td>
</tr>
<tr>
<td>5.14</td>
<td>Problems</td>
<td>170</td>
</tr>
</tbody>
</table>

### 6. Estimation, Planning, and Investment

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Software Size Estimation</td>
<td>174</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Pitfalls and Pratfalls</td>
<td>174</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Software Size Metrics</td>
<td>175</td>
</tr>
<tr>
<td>6.2</td>
<td>Function Points</td>
<td>176</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Fundamentals of FPA</td>
<td>176</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Brief History</td>
<td>176</td>
</tr>
<tr>
<td>6.2.3</td>
<td>Objectives of FPA</td>
<td>177</td>
</tr>
<tr>
<td>6.2.4</td>
<td>Characteristics of Quality FPA</td>
<td>177</td>
</tr>
<tr>
<td>6.3</td>
<td>Five Major Elements of Function Point Counting</td>
<td>177</td>
</tr>
<tr>
<td>6.3.1</td>
<td>EI</td>
<td>177</td>
</tr>
<tr>
<td>6.3.2</td>
<td>EO</td>
<td>178</td>
</tr>
<tr>
<td>6.3.3</td>
<td>EQ</td>
<td>178</td>
</tr>
<tr>
<td>6.3.4</td>
<td>ILF</td>
<td>178</td>
</tr>
<tr>
<td>6.3.5</td>
<td>EIF</td>
<td>179</td>
</tr>
<tr>
<td>6.4</td>
<td>Each Element Can Be Simple, Average, or Complex</td>
<td>179</td>
</tr>
<tr>
<td>6.5</td>
<td>Sizing an Automation Project With FPA</td>
<td>182</td>
</tr>
<tr>
<td>6.5.1</td>
<td>Advantages of Function Point Measurement</td>
<td>183</td>
</tr>
<tr>
<td>6.5.2</td>
<td>Disadvantages of Function Point Measurement</td>
<td>184</td>
</tr>
<tr>
<td>6.5.3</td>
<td>Results Common to FPA</td>
<td>184</td>
</tr>
<tr>
<td>6.5.4</td>
<td>FPA Accuracy</td>
<td>185</td>
</tr>
<tr>
<td>6.6</td>
<td>NCSLOC Metric</td>
<td>186</td>
</tr>
<tr>
<td>6.6.1</td>
<td>Company Statistics</td>
<td>187</td>
</tr>
<tr>
<td>6.6.2</td>
<td>Reuse</td>
<td>187</td>
</tr>
</tbody>
</table>
6.6.3 Wideband Delphi / 189
6.6.4 Disadvantages of SLOC / 190

6.7 Production Planning / 192
6.7.1 Productivity / 192
6.7.2 Mediating Culture / 192
6.7.3 Customer Relations / 193
6.7.4 Centralized Support Functions / 193

6.8 Investment / 195
6.8.1 Cost Estimation Models / 195
6.8.2 COCOMO / 197
6.8.3 Scheduling Tools—PERT, Gantt / 205
6.8.4 Project Manager’s Job / 207

6.9 Example: Apply the Process to a Problem / 208
6.9.1 Prospectus / 208
6.9.2 Measurable Operational Value (MOV) / 209
6.9.3 Requirements Specification / 209
6.9.4 Schedule, Resources, Features—What to Change? / 214

6.10 Additional Problems / 216

7. Design for Trustworthiness 223

7.1 Why Trustworthiness Matters / 224
7.2 Software Reliability Overview / 225
7.3 Design Reviews / 228
7.3.1 Topics for Design Reviews / 229
7.3.2 Modules, Interfaces, and Components / 230
7.3.3 Interfaces / 234
7.3.4 Software Structure Influences Reliability / 236
7.3.5 Components / 238
7.3.6 Open&Closed Principle / 238
7.3.7 The Liskov Substitution Principle / 239
7.3.8 Comparing Object-Oriented Programming With Componentry / 240
7.3.9 Politics of Reuse / 240

7.4 Design Principles / 243
7.4.1 Strong Cohesion / 243
7.4.2 Weak Coupling / 243
7.4.3 Information Hiding / 244
7.4.4 Inheritance / 244
7.4.5 Generalization/Abstraction / 244
7.4.6 Separation of Concerns / 245
7.4.7 Removal of Context / 245

7.5 Documentation / 246
7.6 Design Constraints That Make Software Trustworthy / 248
  7.6.1 Simplify the Design / 248
  7.6.2 Software Fault Tolerance / 249
  7.6.3 Software Rejuvenation / 251
  7.6.4 Hire Good People and Keep Them / 254
  7.6.5 Limit the Language Features Used / 254
  7.6.6 Limit Module Size and Initialize Memory / 255
  7.6.7 Check the Design Stability / 255
  7.6.8 Bound the Execution Domain / 259
  7.6.9 Engineer to Performance Budgets / 260
  7.6.10 Reduce Algorithm Complexity / 263
  7.6.11 Factor and Refactor / 266

7.7 Problems / 268

PART 3 TAKING THE MEASURE OF THE SYSTEM 275

8. Identifying and Managing Risk 277

  8.1 Risk Potential / 278
  8.2 Risk Management Paradigm / 279
  8.3 Functions of Risk Management / 279
  8.4 Risk Analysis / 280
  8.5 Calculating Risk / 282
  8.6 Using Risk Assessment in Project Development: The Spiral Model / 286
  8.7 Containing Risks / 289
    8.7.1 Incomplete and Fuzzy Requirements / 289
    8.7.2 Schedule Too Short / 290
    8.7.3 Not Enough Staff / 291
    8.7.4 Morale of Key Staff Is Poor / 292
    8.7.5 Stakeholders Are Losing Interest / 295
    8.7.6 Untrustworthy Design / 295
    8.7.7 Feature Set Is Not Economically Viable / 296
    8.7.8 Feature Set Is Too Large / 296
    8.7.9 Technology Is Immature / 296
    8.7.10 Late Planned Deliveries of Hardware and Operating System / 298
  8.8 Manage the Cost Risk to Avoid Outsourcing / 299
    8.8.1 Technology Selection / 300
    8.8.2 Tools / 300
    8.8.3 Software Manufacturing / 300
    8.8.4 Integration, Reliability, and Stress Testing / 301
    8.8.5 Computer Facilities / 301
    8.8.6 Human Interaction Design and Documentation / 301
11. Testing and Configuration Management

11.1 The Price of Quality / 373
  11.1.1 Unit Testing / 373
  11.1.2 Integration Testing / 373
  11.1.3 System Testing / 373
  11.1.4 Reliability Testing / 374
  11.1.5 Stress Testing / 374

11.2 Robust Testing / 374
  11.2.1 Robust Design / 374
  11.2.2 Prototypes / 375
  11.2.3 Identify Expected Results / 375
  11.2.4 Orthogonal Array Test Sets (OATS) / 376

11.3 Testing Techniques / 376
  11.3.1 One-Factor-at-a-Time / 377
  11.3.2 Exhaustive / 377
  11.3.3 Deductive Analytical Method / 377
  11.3.4 Random/Intuitive Method / 377
  11.3.5 Orthogonal Array-Based Method / 377
  11.3.6 Defect Analysis / 378

11.4 Case Study: The Case of the Impossible Overtime / 379

11.5 Cooperative Testing / 380

11.6 Graphic Footprint / 382

11.7 Testing Strategy / 384
  11.7.1 Test Incrementally / 384
  11.7.2 Test Under No-Load / 384
  11.7.3 Test Under Expected-Load / 384
  11.7.4 Test Under Heavy-Load / 384
  11.7.5 Test Under Overload / 385
  11.7.6 Reject Insufficiently Tested Code / 385
  11.7.7 Diabolic Testing / 385
  11.7.8 Reliability Tests / 385
  11.7.9 Footprint / 385
  11.7.10 Regression Tests / 385

11.8 Software Hot Spots / 386

11.9 Software Manufacturing Defined / 392

11.10 Configuration Management / 393

11.11 Outsourcing / 398
  11.11.1 Test Models / 398
  11.11.2 Faster Iteration / 400
  11.11.3 Meaningful Test Process Metrics / 400

11.12 Problems / 400
12. The Final Project: By Students, For Students 404

12.1 How to Make the Course Work for You / 404
12.2 Sample Call for Projects / 405
12.3 A Real Student Project / 407
12.4 The Rest of the Story / 428
12.5 Our Hope / 428

INDEX 429
Preface

This book advances the idea that standard principles of good engineering, which have been honed through meticulous application, examination, revision, and refinement, should be practiced in creating trustworthy software. It provides the student/practitioner with structured experiences that teach the critical engineering skills needed to build reliable software products. Good practices are illustrated with case studies and elicited through practical projects that apply stresses common in the business world after which student responses and their logical consequences are examined. Quantitative analysis is applied to software engineering principles. The course based on this book is managed in a style modeled on industrial software development.

Lawrence Bernstein teaches in the Computer Science Department at Stevens Institute of Technology in Hoboken, NJ. For more than 30 years, until his retirement in 1996, he was a leading software executive and software project manager at Bell Laboratories. He has more than 40 years of experience in developing, managing, and teaching software. He saw that developers too often do not want to think about the downstream consequences of what they do. To the extent that we can teach a code of ethics, we can raise the quality of practice and reduce risks. This book confronts both new and experienced developers with the need for critical thinking during software development and for making those engineering tradeoffs that will make systems safe, secure, and reliable.

I HEAR AND I FORGET; I SEE AND I REMEMBER; I DO AND I UNDERSTAND (CONFUCIUS)

The aim is to give the student/practitioner the intellectual and teamwork skills needed to make practical engineering tradeoffs during product realization.
Too often, novice computer scientists jump into software development ignorant of proven tools, processes, and theory. Passing mention in a distant course lecture hardly suffices to impress how critical to project success are such practical tradeoffs. They learn from harsh on-the-job experience just how important it is to plan, measure, and assess each stage of development. This book captures these experiences and describes them in the context of real-life examples. Rather than preaching how important software engineering is and dwelling on the “Best Current Practices,” the emphasis is instead on problem analysis, fitting the software engineering structure to the problem, and producing products that are on schedule, within budget, and satisfactory to the customer. The concepts of simplification, trustworthiness, risk assessment, and architecture are stressed.

Software engineering, as opposed to hacking, is a key feature of this text. Software engineering is the ability to make judgments based on measurements to structure and monitor the development of a software product. Engineering implies that limitations face the developer. In software, the issue of balancing feature content, schedule time, system performance, and cost face the project manager every day.

**PERSPECTIVE STUDY**

Two Stevens Institute of Technology professors examined the effects of the course described in this book. They conducted a perspective study of 150 students who took the course over a period of 3 years. Before beginning the course, naïve students were questioned concerning the use of software engineering methods and the acceptance of constraints. Approximately 60% voiced acceptance, which might have been colored by lack of experience and malleability, but 40% vigorously opposed the constraints of the process, perhaps dazzled by the glories of hacking. By the completion of the two-semester course, most students were competent to function in design teams of eight to ten people and to consistently apply the rigorous standards of engineering to software. Of those students initially opposed, only 10% remained opposed and opted to work in fields requiring small projects that would suit their solitary natures. A significant measure of the teaching technique’s success was the ability of the students to find jobs in large companies during difficult economic times because of successful interviews. The students had mature answers to situations posed by interviewers and reported that the course gave them a differential advantage.

**PREREQUISITES**

The idea of project-based development drives the course. Assignments are structured so that fundamental software engineering practices are provided
exactly when the need arises. The student begins to appreciate the need to go beyond the boundary of the small one-, two-, or three-person assignments that were used to teach programming. This book expects the reader to know programming and expands that perspective to include the real-world problems facing projects requiring 10 to 25 people a year. Larger projects exist, but they are not emphasized in this text.

It is important that software project teams consist of eight to ten students working together for one, one and a half, or two semesters. Some students and faculty object to this large size and long duration, but smaller projects do not help students learn to deal with the problems of communication, priority setting, risk assessment, and dealing with different personalities. On smaller projects, students tend to self-select people who are like themselves and thereby avoid conflicts.

**METHOD**

Students attempt a short initial project to experience the real-world environment. Then they work on a second project, longer in duration, that produces a useful software product employing the best current software practices. The practices show how the problems faced on the first project can be handled in a systematic way.

Many computer science, information technology, computer engineering, and software engineering courses require students to build a software product, but building a system out of individual little products is brushed by theoretical steps only, with little emphasis on how to manage the product realization process. The complexity of project interactions is not emphasized. In this course, it is actively experienced.

Regrettably, students often are taught how to use tools but not how to make critical judgments about the appropriateness of when and where to use them. There are many software engineering tools. Projects can fail by running after the latest fad and therefore having no stability in their development environment. At the other extreme, projects become obsolete or too expensive when new tools are fearfully shunned. How a software engineer decides whether to adopt a tool is an important part of a software engineering education. UNIX and Linux operating systems, the C, C++, and JAVA languages, software tools that find memory leaks, fault tolerance, stress testing, regression testing, configuration management, change control, informal project meetings, prototyping, visual programming, and the Spiral Model constitute a brief litany of some that are successes. This book explains these and other tools and then provides the criteria for deciding whether to use them. Students are then competent to solve the inevitable problems they face developing a software product.

Supplemental materials are used in this course such as videotapes and current news summaries. These materials can be accessed through the publisher’s website associated with this book.
AUDIENCE

Software professionals will refer to this book as a guide during their real-life projects. Experienced project managers share the common experience of failure but do not often have the luxury of examining the causes that are common to failures. The novice does not yet have this experience. This book follows a logical path of increasing software engineering appreciation by the developer in contrast to the common style of following a prescribed theoretical software development process. It is not a complete compendium of all development theories, but it serves as the foundation to the entire quantitative software engineering series.

The emphasis of this book is on quantitative software engineering decision making. It is geared toward the student who wants to know the “why” as well as the “how.” Successful completion of the course described in this book makes a software engineer valuable to a company both as a team member and as a manager of outsourced developments.

Overall, it is shocking how few of the available tools or software engineering practices are actually used in software project development. With marketing people currently in ascendance and controlling development budgets, there is no comprehension of the need to invest in tools and skills development. Software professionals under marketing pressures have neither the resources to grow nor to argue their cases effectively. Hope lies in the freshly educated engineers emerging from colleges and universities who are fluent in mathematics and programming and convinced by their own experience of the utility of new methods and tools. The systematic adoption of software engineering practices in industry requires that an appreciation of its importance be conveyed to computer science students.

THE SOFTWARE CRISIS

In the fall of 1968, NATO convened a meeting to confront a crisis that did not make the headlines—“the software crisis.” Experts from a dozen countries, representing industrial laboratories as well as universities, met in Garmisch, Germany, to grapple with two basic questions that are still with us: Why is it so hard to produce functionally correct and reliable software that meets both users’ needs and performance requirements and still comes in on time and within a projected budget? Where should software producers be looking for solutions? The answers revolved around “software engineering,” a term coined by the organizers of the Garmisch conference—somewhat controversial at the time, and wholly inspirational—to focus on a missing discipline with the potential to resolve the crisis. Software production should be “industrialized.” Systems should be built from reusable components. Development processes should be more systematic and predictable. The people who design, produce, and maintain software should be better equipped to do what engineers do: Make things that work to meet someone else’s needs.
There have been significant advances. Programmers are better now, most code is written in high-level languages, better tools exist, development is done online, better design models exist, and standards have been established in some key areas. Several recently developed or recently successful innovations include object-oriented programming, client/server and thin client applications, application programming interfaces, graphical user interfaces and development tools, prototyping, and source-level debugging. Components exist and are widely used, which belies the common wisdom that software components are rare. The C libraries with all their richness of fundamental or atomic functions provide 20% reuse in most industrial strength UNIX-based applications. Software that makes a library of graphical elements, or a text-processing tool, available to multiple applications, with a major reservation, is also in wide use. IBM and others are using software libraries providing generic fault avoidance and recovery.

The industry has been much less successful in creating larger reusable components or “bricks” than in reusing some kinds of “mortar” to hold programs together—such as UNIX pipes and filters or object-oriented programming techniques. There have been significant advances in software process—in areas such as incremental development, inspection, change control, and testing—and in software tools such as test drivers, profilers, and configuration management. These process advances have helped software producers gain ground on the elusive goal: timely, cost-effective development of reliable, user-friendly software. Yet as far as software research and development have come since 1968, there is still a long way to go, in part because progress in the field tends to have an amplified effect on rising expectations. The central problems discussed in Garmisch persist. Software design and production still do not resemble engineered industrial processes. Maybe they never will. Reuse is still an issue, more promise than practice. Complexity has been compounded by networking and distributed applications.

Like the problems, some of the most promising ideas and techniques have been around for a while. The idea of cleanly separating design intent from the method of implementation is a powerful one with a lot of mileage left in it. Modularity based on the Parnas concept of information hiding has been a key to progress in software and will continue to be applied broadly and on many levels. The idea of layered architectures—with software on one layer connecting to software on another—is a proven aid to interoperability in network systems and is working its way through all sorts of applications. Browsers are a familiar kind of layer that users see; platforms and middleware are typical uses of layered software behind the scenes. Domain engineering embodies key insights for large-scale software reuse: Think in terms of families of software components within a domain, and determine what common elements can be reused in several generations; develop reusable components and establish a process for using them to build new applications—new product line members—rapidly, efficiently, and with confidence in their reliability and performance. Experiences with domain engineering indicate that it could help software producers realize the promise of reuse.
All approaches to problems in software continue to benefit from program notation and formal methods, remedies that come from software’s mathematical roots. Programming languages have long provided precise notations for specifying computations. There is movement toward the use of specifications to capture inter-relationships as well, which makes it easier to reason about the behavior of complex software such as distributed systems.

Complementary work emphasizes the human side of software engineering. An early insight repeatedly confirmed by experience is that development problems have as much to do with people as with technology. In programming languages, a trend exists toward more intuitive notations. Discovery tools such as data visualization can make it easier for members of a development team to get their heads around millions of lines of code. Another prime example of the human focus is the patterns discipline (sometimes called the patterns “movement” to distinguish it as an idea taking hold in a community). Developers are recording design insights and experience in “patterns”—more a literary form than a kind of documentation—each pattern capturing “a solution to a problem in a context.” Collections of patterns that work together are said to form a “pattern language.” The patterns discipline assumes that most knowledge needed to build and maintain a reliable system already resides in the development community. But people tend to move around, taking expertise with them, and projects can go on for a long time. Many developers must work with code that has been around longer than they have. That fact produces questions: What was the software architect’s original intent? What lessons did the developers learn? How can I get inside the experts’ minds?

It will be important to experiment with various software tools and practices to see how much better the technical solutions fare in concert with approaches that focus on the people who build the software systems.

WHO CAN PROFIT FROM THIS BOOK

This book presents the basic skills needed to apply these software technologies to the realization of software products on time, within budget, and with known quality. It is especially useful for those who must produce trustworthy software systems. It is geared toward several kinds of readers:

• The formally educated computer professional who aspires to a managerial career and wants comprehensive hands-on knowledge in the skills needed to identify customer requirements, develop software designs, manage a software development team, and evaluate the resulting software product relative to customer specifications.

• The formally educated computer professional who wants to remain an individual contributor, yet wants a solid foundation in the practical application of computer science technology to the realization of software products.
• The computer professional whose educational background is not in computer science or computer engineering, but who has learned software skills on the job and who now wants to begin to understand software engineering.

• The systems engineer and software project manager who wants to understand software engineering technology.

• The venture capitalist who wants to assess the likelihood of a software company’s success.

• The CEO who finds software fundamental to the company’s products and services.

The book may be used for a one- or two-semester undergraduate course in software engineering. The one-semester course should meet 3 hours/week for 14 weeks with weekly project meetings outside of class. Class meetings are for project presentations, presentation of case histories and technologies, and discussions. Each project presents a requirements review, an architecture and design review, a code and test review, and a final product demonstration. Weekly progress reports are suggested. Monthly updates to the development plans are helpful. If the instructor wishes to have the project meetings during scheduled class time and to provide time for a larger, more realistic and richer project experience, then a two-semester sequence such as that used at Stevens is recommended. Intensive 6-day professional short courses can be tailored to specific needs.

THE ISSUE OF TRUSTWORTHINESS

Software trustworthiness, which refers to the attributes of reliability, security, and safety, is the next major area in which academia and industry must focus. Software trustworthiness is critical for medical devices, plant control systems, and weapon systems; for management of sensitive records; for critical infrastructure; for new accounting requirements; and for dependable cybersecurity. As practitioners of such a pervasive part of so many lives, we have a serious ethical responsibility. This course will help make you ready to assume your responsibility for trustworthiness in our young profession. Additional information is available on the publisher’s website at ftp://ftp.wiley.com/public/sci_tech_med/trustworthy_systems.

Lawrence Bernstein  
Industry Research Professor  
Stevens Institute of Technology  
Hoboken, New Jersey

C. M. Yuhas  
Freelance Writer  
Short Hills, New Jersey
Acknowledgment

There is an Italian saying, “All roads lead to Rome.” In writing this textbook, we became acutely aware that for much of the innovation in our field, all mental roads lead to Barry Boehm. We are grateful for his inspiration, his advice, and his friendship.
Part 1

Getting Started