# Making Software Engineering Happen

## A Guide for Instituting the Technology

Roger S. Pressman, Ph.D.

# Making Software Engineering Happen

## A Guide for Instituting the Technology

Roger S. Pressman, Ph.D.

*President, R.S. Pressman & Associates, Inc.*

*Adjunct Professor of Computer Engineering*
*University of Bridgeport*

Prentice Hall, Englewood Cliffs, New Jersey 07632

*Library of Congress Cataloging-in-Publication Data*

PRESSMAN, ROGER S. Making software engineering happen.

Bibliography.
Includes index.
1. Computer software—Development. I. Title.
QA76.76.D47P74 1988 005.1 87-17438
ISBN 0-13-547738-7

Cover design: Roger S. Pressman
Manufacturing buyer: Paula Benevento



Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-547738-7 025

PRENTICE-HALL INTERNATIONAL (UK) LIMITED, *London*
PRENTICE-HALL OF AUSTRALIA PTY. LIMITED, *Sydney*
PRENTICE-HALL CANADA INC., *Toronto*
PRENTICE-HALL HISPANOAMERICANA, S.A., *Mexico*
PRENTICE-HALL OF INDIA PRIVATE LIMITED, *New Delhi*
PRENTICE-HALL OF JAPAN, INC., *Tokyo*
SIMON & SCHUSTER ASIA PTE. LTD., *Singapore*
EDITORA PRENTICE-HALL DO BRASIL, LTDA., *Rio de Janeiro*

# PREFACE

Today, managers in companies of every kind recognize that computer software is an essential part of their business. Software provides information for business decision making, it provides data that enable engineers and scientists to effectively apply technology, it becomes the differentiating characteristic in products and systems of every description. But these beneficial traits are often outweighed by something else: software remains a bottleneck. We continue to build computer programs for the 1990s using development approaches that have remained virtually unchanged since the 1970s. Something has to be done.

There is no simple solution to a "software crisis" that has been brewing for thirty years. But one thing is clear: software can be developed more productively with significantly higher quality when a disciplined approach is taken—an approach called *software engineering*.

Software engineering encompasses procedures, methods and tools for the analysis, design, implementation and testing of computer software. First introduced in the early 1970s, software engineering has evolved into a multi-disciplinary field that encompasses elements of computer science, management science and pragmatic engineering. Although few people disagree with software engineering philosophy, it has not been adopted as widely as one might like. The problem is *making it happen!*

Software engineering represents a significant departure from the art form that we call "computer programming." Although the objectives of software engineering and programming are the same, it is the difference in approach that creates problems. Software engineering requires a software development organization to undergo technology transition. This leads to changes—changes in procedures, changes in

the way people perceive their work and the manner in which their work is measured, changes in the methods and tools that are used to build the products and systems that lead to successful software. Unfortunately, changes are never easy!

For the past ten years I have had the opportunity to help a number of large companies implement software engineering practices. The job is difficult and rarely goes as smoothly as one might like—but when it succeeds, the results are profound. Software projects are more likely to be completed on time. Communication between all constituencies involved in software development is improved. The level of confusion and chaos that is often prevalent for large software projects is reduced substantially. The number of errors encountered by the customer drops. The credibility of the "software organization" increases. And management has one less problem to worry about.

But everything is not sweetness and light. Many companies attempt to implement software engineering practice and give up in frustration. Others do it half way and never see the benefits noted above. Still others take a heavy handed approach that results in open rebellion among technical staff and managers and subsequent loss of morale.

*Making Software Engineering Happen* has been written to help managers who recognize the need for a more disciplined approach to software development, but don't really know how to start the transition to software engineering. The book has been written because too many companies go about software engineering implementation in the wrong way, using the wrong people, who focus on the wrong problems.

*Making Software Engineering Happen* is not intended to be a comprehensive tutorial on software engineering. Many good textbooks already exist to fill this need. Rather, the book has been designed as a manager's guide that will enable you to put software engineering into place—to supplement your own base of experience and provide a road map for those who have not managed the implementation of software engineering methodology or have not yet completed the job.

Early chapters of the book present an overview of software engineering (Chapter 1) and examine software engineering implementation and the impact of cultural change that results as a consequence of this new technology (Chapter 2). Chapter 3 introduces a "life cycle" for

the implementation of software engineering and CASE—that is, the steps required to make software engineering happen. The model defines strategies for assessment, education, technology selection, installation and evaluation.

Chapters 4 through 8 describe each step in the software engineering implementation life cycle in detail. Among the many topics presented are qualitative and quantitative assessment, education strategies, methods and tools selection checklists, and guidelines for methodology installation and evaluation. These chapters are supplemented with a number of appendices. Appendix B is of particular importance because it presents "quasi-expert system" guidance for conducting a *Software Engineering Audit* —a procedure that enables you to assess the current state of software development practice within your organization.

Chapter 9 recommends a strategy for software engineering implementation. A work breakdown structure defines all steps in the strategy, a typical implementation schedule is established, and important management issues are discussed.

It would be dishonest to say that the approach described in this book is fool proof. Dedicated people, not books, make software engineering happen. But the activities, guidelines, suggestions and approach described within these pages can be useful to managers and others who are chartered with the responsibility for making software engineering become a reality in their organizations.

The ideas contained in this book have evolved from many sources: an off-the-cuff remark by a client during a meeting; a criticism from a student in a classroom; a comment in a letter from a colleague; an article in a trade journal; an insight in a textbook. All of these sources have contributed to *Making Software Engineering Happen.* Most important, however, have been the many months spent with industry clients attempting to solve real-life problems associated with software engineering implementation.

Many of the ideas presented in these pages first appeared in a consultant's guide which I developed for Ernst & Whinney. My heartfelt thanks to Clint Alston and his colleagues at Ernst & Whinney for providing me with the incentive to "put it into writing" and allowing me to reuse important segments of their consultant's guide in this book.

*Roger S. Pressman*

# CONTENTS

**PREFACE** ... xi

**1 INTRODUCTION** ... 1

1.1 Technology Implementation ... 3
1.2 An Overview of Software Engineering ... 3
1.2.1 Software Engineering—A Generic View ... 4
1.2.2 Software Engineering Paradigms ... 5
1.2.3 The Classic Life Cycle ... 5
1.2.4 Prototyping ... 7
1.2.5 Fourth Generation Techniques ... 9
1.2.6 Combining Paradigms ... 10
1.3 Computer-Aided Software Engineering (CASE) ... 11
1.4 Software Quality Assurance and Software Configuration Management ... 12
1.5 Organizing for Software Engineering ... 13
1.5.1 The Software Engineering Project Team ...14
1.5.2 Lines of Communication ... 15
1.6 Summary ... 16

**2 THE CHALLENGE OF TECHNOLOGICAL CHANGE** ... 19

2.1 The Challenge of Cultural Change ... 20
2.2 People and Change ... 22
2.3 Getting Staff Commitment ... 25
2.4 Understanding Customer/User Concerns ... 26
2.5 Getting the Message Across ... 27
2.6 Selecting a "Local Champion" ... 30
2.7 Summary ... 31

**3 SOFTWARE ENGINEERING IMPLEMENTATION LIFE CYCLE ... 33**

3.1 The Implementation Life Cycle ... 34
3.1.1 The Assessment Step ... 34
3.1.2 The Education Step ... 35
3.1.3 The Selection Step ... 37
3.1.4 The Installation Step ... 38
3.1.5 The Evaluation Step ... 39
3.2 Implementation Process Flow ... 41
3.3 Factors that Affect the Life Cycle ... 42
3.4 Summary ... 43

**4 ASSESSMENT ... 45**

4.1 The Software Engineering Audit ... 46
4.1.1 Subject Areas Covered in the Audit ...46
4.1.2 Who Conducts the Audit? ... 47
4.2 Qualitative Assessment ... 48
4.3 Quantitative Assessment ... 52
4.3.1 Line of Code Measures ... 53
4.3.2 Function Point Measures ...53
4.3.3 Guidelines for Quantitative Assessment ... 54
4.3.4 Reporting Quantitative Findings ...56
4.4 The Assessment Report ... 57
4.4.1 Guidelines for Preparation ... 60
4.4.2 Format and Content ... 61
4.4.3 Presenting Your Findings to Management ... 63
4.5 The Transition Plan ... 65
4.5.1 Guidelines for Preparation ... 66
4.5.2 Format and Content ... 66
4.6 Summary ...69

**5 EDUCATION ... 71**

5.1 Understanding Educational Needs ... 72
5.1.1 Generic Methods and Concepts ... 72
5.1.2 Tools ... 74
5.1.3 General Business Practice ... 75
5.2 Establishing a Training Strategy ... 75
5.3 Guidelines for Short Courses and Seminars ... 76
*Software Engineering: A Business Perspective* ... 78
*Management Course in Software Engineering* ... 79
*Software Engineering Methodology Course* ... 80
5.4 Sources of Supplementary Courseware ... 81
5.4.1 Supplementary Literature ... 81

5.4.2 Supplementary Courseware ... 82
5.5 Summary ... 83

**6 SELECTION ... 85**

6.1 The Selection Process ... 86
6.2 Generic Guidelines for Selection ... 88
6.3 Productivity and Quality Issues ... 90
6.4 Selection Criteria ... 90
6.4.1 Procedures ... 91
6.4.2 Methods ... 93
6.4.3 Tools ... 94
6.5 A CASE Tools Checklist ... 96
6.6 Justification ... 99
6.6.1 A Simple Justification Model ... 100
6.6.2 Intangibles that Affect Justification ... 102
6.7 Justifying CASE ... 102
6.8 Summary ... 104

**7 INSTALLATION ... 105**

7.1 Technology Transition ... 106
7.2 Transition Problems ... 107
7.2.1 Education Problems ... 108
7.2.2 Tools Problems ... 108
7.2.3 Methods Problems ... 110
7.2.4 Procedural Problems ... 110
7.3 Instituting Software Engineering Procedures and Methods ... 111
7.3.1 Project Tracking and Control ... 112
7.3.2 Analysis Procedures and Methods ... 113
7.3.3 Preliminary and Detail Design ... 114
7.3.4 Code Generation ... 116
7.3.5 Testing ... 117
7.3.6 Documentation and Deliverables ... 119
7.3.7 Software Configuration Management ... 121
7.4 Installing Software Engineering Tools ... 122
7.4.1 Installing Coding Tools ... 123
7.4.2 Installing CASE Tools ... 125
7.4.3 Installing Desk-Top Publishing ... 127
7.5 Software Quality Assurance Procedures ... 129
7.6 Developing Standards and Procedures ... 130
7.7 How to Get Started ... 133
7.7.1 Evaluating a Pilot Project ... 136
7.7.2 Propagating Software Engineering to Other Projects ... 137
7.8 Summary ... 137

**8 EVALUATION** ... 139

8.1 Measuring the Success of Installation ... 140
8.2 Measuring Productivity and Quality ... 140
8.3 Tuning the Approach ... 141
8.4 Justification for Management ... 143
8.5 Summary ... 144

**9 AN IMPLEMENTATION STRATEGY** ... 145

9.1 A Work Breakdown Structure ... 146
  9.1.1 Preliminaries ... 146
  9.1.2 Assessment Tasks ... 147
  9.1.3 Education Tasks ... 150
  9.1.4 Selection Tasks ... 152
  9.1.5 Installation and Evaluation Tasks ... 154
9.2 A Schedule for Software Engineering Implementation ... 157
  9.2.1 Task Sequences and Interdependencies ... 158
  9.2.2 Phased Implementation ... 165
9.3 Strategic Issues ... 166
9.4 Summary ... 168

**EPILOGUE** ... 169

**APPENDICES**

**A** Self-Assessment for Software Engineering Practice ... 171

**B** Software Engineering Audit ... 187

**B.1** Software Engineering Audit Questionnaire ... 189

**B.2** Audit Comments, Inferences, and Follow-up Questions ... 197

**C** Spreadsheet Model for Assessment and Evaluation ... 227

**D** An Annotated Outline for the Transition Plan ... 233

**E** Software Engineering Bibliography ... 239

**INDEX** ... 253

# 1

# INTRODUCTION

Software and software based systems are important to your business. Whether software is embodied as an information system for management decision making, a support system for manufacturing, or an embedded element of a computer-based product, it serves to distinguish one company from another.

Today, software development technology is undergoing change. Companies that could once develop computer-based systems using an informal approach find that a more disciplined and controlled methodology is essential to meet budgeted costs, maintain required schedules, and achieve desired quality. People who once developed programs with little more than a programming language compiler are now using a suite of automated tools that span all important tasks in software development. But why is all this change necessary?

The answer to the above question is pivotal to an understanding of the challenges that face any company that attempts to change the

manner in which software is developed. Techniques that were applied successfully in the 1960s, 1970s, and in some cases, the 1980s, do not work today because:

1. software has become larger and much more complex for applications of all kinds;
2. the success or failure of a software development project often has a traceable impact on the bottom line;
3. customers demand more rapid response to their needs—the business climate requires fast reaction;
4. computer-based systems demand interdisciplinary skills—that is, as systems have grown, software people must interact with many other constituencies;
5. good technical people—the most important software development resource—are a scarce and expensive commodity; and
6. senior management is more aware of software and is less likely to be "snowed" by excuses and jargon that were often voiced during the early days of computing.

A discipline called *software engineering* has been adopted by many companies as a response to the changing systems environment. Software engineering encompasses a variety of technical methods, a set of management procedures, and a suite of automated tools (often called CASE—*computer-aided software engineering*) that enhance our ability to build effective computer-based systems. The collection of methods, procedures and tools that comprise software engineering represent a fundamental departure from the "old ways." The *ad hoc* style of the past must give way to a more controlled approach.

Companies with a long established software development culture often find it difficult to adopt new methods, tools, and procedures. Yet, management at these companies recognizes that the old culture is ineffective and in some cases detrimental to the overall strategic objectives.

When viewed from the inside, the need for modern software engineering practice puts many managers in a quandary. The current approach gets the job done, but costs are unpredictable, schedules tend to slip, and overall quality is sometimes less than adequate. Yet, the current approach to building systems (with all of its failings) is reasonably well accepted by middle managers and technical practitioners alike. When new methods, procedures and CASE tools are

suggested, old ways are threatened, job functions may change, and new things must be learned—the boat has been rocked. Proposed changes may be resisted by the very people who could benefit the most from them. To institute software engineering practice, a manager must be willing to institute change— and change is never easy.

## 1.1 TECHNOLOGY IMPLEMENTATION

As we begin our discussion of software engineering implementation, it is important to understand that in the context of this book, the term *implementation* refers not to the creation of systems or software, but to the cultivation of a new technology (software engineering) within a human organization. The remaining chapters of this book consider many aspects of implementation. For now, we consider its most general attributes:

**Technology assessment.** Before later stages of implementation can commence, it is necessary to assess the current state of software/system development practice. Problem areas and strengths are isolated; recommendations for change are made, and a *Transition Plan* is developed.

**Technology transfer.** Once the software development organization has been assessed and a plan of action defined, a set of activities that support methodology installation should be conducted. These activities include education of both management and technical staff, selection and installation of appropriate methods and tools, and definition of evaluation criteria.

**Technology evaluation.** Implementation does not end with the transfer of software engineering technology. Rather, the application of the technology is evaluated (both qualitatively and quantitatively). Procedures, methods and tools are then tuned to optimize their application.

## 1.2 AN OVERVIEW OF SOFTWARE ENGINEERING

Software engineering is a software development discipline that can be applied to applications of all kinds. The discipline encompasses three major elements:

1. *Procedures* for planning, controlling, tracking and assuring technical activities during software development ;

2. *Methods* that are applied by technical staff during the analysis, design, implementation, testing and maintenance of software;
3. *Tools* that create a hardware and software support environment that complements software engineering methods and procedures.

Pressman* provides a concise description of the interrelationship of these elements:

> Software engineering methods provide the technical "how to's" for building software. Methods encompass a broad array of tasks that include: project planning and estimation, system and software requirements analysis, design of data structure, program architecture and algorithm procedure, coding, testing and maintenance. Methods for software engineering often introduce a special language-oriented or graphical notation and introduce a set of criteria for software quality.
>
> Software engineering tools provide automated or semi-automated support for methods. Today, tools exist to support each of the methods noted above. When tools are integrated so that information created by one tool can be used by another, a system for the support of software development, called *computer-aided software engineering* (CASE), is established. CASE combines software and hardware to create a software engineering environment that is analogous to CAD/CAE (computer aided design/engineering) for hardware.
>
> Software engineering procedures are the glue that holds the methods and tools together and enables rational and timely development of computer software. Procedures define the sequence in which methods [and tools] will be applied, the deliverables (documents, reports, forms, etc.) that are required, the controls that help assure quality and coordinate change, and the milestones that enable software managers to assess progress.

A specific characterization of software engineering—at both philosophical and pragmatic levels—will vary among companies and application areas. However, the three elements noted above will always be present.

### 1.2.1 Software Engineering—A Generic View

Software engineering can be characterized in a number of different ways. When the chronology of the software development process is examined, software engineering flows through three generic phases: *definition, development* and *maintenance*. The phases are often modeled using one or more *paradigms* that describe the individual steps

* Pressman, R.S., *Software Engineering: A Practitioner's Approach*, second edition, McGraw-Hill, 1987.

associated with each phase. When the pragmatic elements of software engineering are considered, the discipline may be viewed as a set of procedures, methods and tools. Finally, when the end-product (working, maintainable software) is the focus, software engineering may be viewed as a set of *deliverables* that encompass documents, programs and data. In actuality, each of these views overlaps the others.

### 1.2.2 Software Engineering Paradigms

A number of different procedural approaches, called *paradigms*, to software engineering have been proposed over the past decade. A paradigm describes the manner in which procedures are characterized, methods are applied, and tools are used. A manager with responsibility for making software engineering happen should never become dogmatic about his or her "favorite" paradigm. That is, the appropriate paradigm for the application of software engineering should be based on the characteristics of the computer-based system to be developed or maintained, the background and knowledge of developers and customers, and available software development resources.

### 1.2.3 The Classic Life Cycle

Figure 1.1 illustrates the classic life cycle paradigm for software engineering. Sometimes called the "waterfall model," this paradigm describes the process of software development as a series of sequential steps in which information developed in one step is built upon to create information for the next step. The following activities occur during the life cycle paradigm:

**Planning.** The business requirements that have created a need for new or modified software are examined. Top level customer requirements are identified, functional and system interfaces are defined and the relation of this software to overall business function is established. In many application areas, this step is expanded to include system engineering as it relates to computer software.

**Analysis.** Detailed requirements necessary to define the function and performance of the software are defined. In addition, the information domain for the system is analyzed to identify data flow characteristics, key data objects and overall data structure.

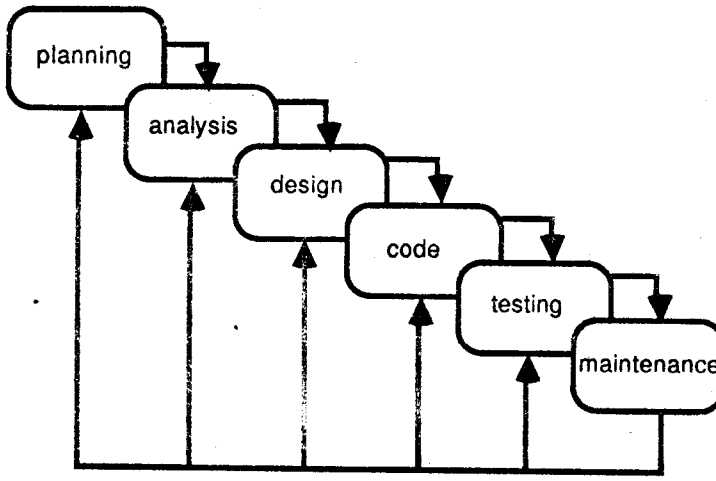


**Figure 1.1** The Classic Life Cycle

**Design.** Detailed requirements are translated into a series of system representations that depict how the software will be constructed. The design encompasses a description of program structure, data structure and, at lower levels, detailed procedural descriptions of the software.
**Code.** Design must be translated into a machine executable form. The coding step accomplishes this translation through the use of conventional programming languages (e.g., C, Ada, Pascal) or so-called *fourth generation languages* (e.g., NOMAD, INTELLECT).
**Testing.** Testing is a multi-step activity that serves to verify that each software component properly performs its required function and validates that the system as a whole meets overall customer requirements.
**Maintenance.** Maintenance is actually the re-application of each of the preceding activities for existing software. The re-application may be required to correct an error in the original software, to adapt the software to changes in its external environment (e.g., new hardware, operating system, etc.), or to provide enhancement to function or performance requested by the customer.

The classic life cycle paradigm is the most widely used approach to software engineering. It leads to systematic, rational software development, but like any generic model, the life cycle paradigm can be problematic for the following reasons:

1. The rigid sequential flow of the model is rarely encountered in

real life. Iteration can, and does, occur, causing the sequence of steps to become muddled.

2. It is often difficult for the customer to provide a detailed specification of what is required early in the process. Yet, this model requires a definite specification as a necessary building block for subsequent steps.

3. Much time can pass before any operational elements of the system are available for customer evaluation. If a major error in implementation is made, it may not be uncovered until much time has passed.

Do these potential problems mean that the life cycle paradigm should be avoided? Absolutely not! They do mean, however, that the application of this software engineering paradigm must be carefully managed to ensure successful results.

### 1.2.4 Prototyping

*Prototyping* is a software engineering paradigm that has been proposed to circumvent some of the problems that are inherent in the classic life cycle. Illustrated schematically in Figure 1.2, prototyping is a modeling process that moves the developer and customer toward a

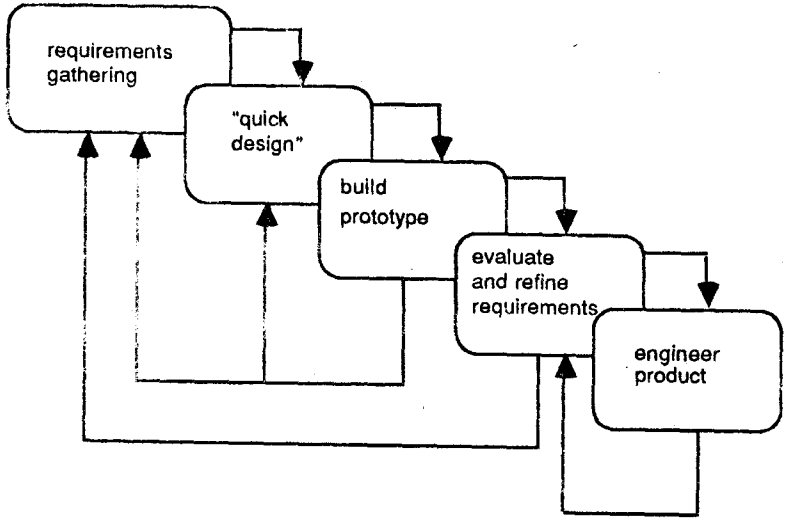


**Figure 1.2** Prototyping