Preface to the Third Edition

An introductory course in Software Engineering remains one of the hardest subjects to teach. Much of the difficulty stems from the fact that Software Engineering is a very wide field which includes a wide range of topics. Consequently, what should be the focus of an introductory course remains a challenge with many possible viewpoints.

This third edition of the book approaches the problem from the perspective of what skills a student should possess after the introductory course, particularly if it may be the only course on software engineering in the student's program. The goal of this third edition is to impart to the student knowledge and skills that are needed to successfully execute a project of a few person-months by employing proper practices and techniques. Incidently, a vast majority of the projects executed in the industry today are of this scope—executed by a small team over a few months. Another objective of the book is to lay the foundation for the student for advanced studies in Software Engineering.

Executing any software project requires skills in two key dimensions—engineering and project management. While engineering deals with issues of architecture, design, coding, testing, etc., project management deals with planning, monitoring, risk management, etc. Consequently, this book focuses on these two dimensions, and for key tasks in each, discusses concepts and techniques that can be applied effectively on projects.

The focus of the book remains as the first course in software engineering, and it retains its character of having a running case study with most of the outputs available. This edition draws upon my experience during my sabbaticals with two software companies—Infosys Technologies and Microsoft Corporation—and my practice-oriented book *Software Project Management in Practice* (Addison-Wesley, 2002) to bring, in addition to the concepts, more elements of how these concepts are actually applied in practice.

In this edition, new material has been added on current practices, out-

dated material has been removed, and discussion has been sharpened. The following key additions have been made:

- In "Software Process" a discussion on the timeboxing model for iterative development and on inspection process
- In "Requirements Analysis and Specification" a description of Use Cases
- A new chapter on "Software Architecture"
- In "Project Planning" some practical techniques for estimation, scheduling, tracking, risk management, etc.
- In "Object Oriented Design", discussion on UML and on concepts like cohesion, coupling, and open-closed principle
- In "Coding" many additions have been made. These include refactoring, test driven development, and pair programming, as well as a discussion on common coding defects, coding standards, and some useful coding practices.
- In "Testing" a discussion on pair-wise testing as an approach for functional testing, defect tracking, and defect analysis and prevention

In addition to the old case study, a new case study has been added. Various work products of the case studies, including the SRS, architecture document, project plan, design document, code, and test plan, have been made available through the Web site.

A Web site has been created for this edition. In addition to outputs of the case studies, implementations of some of some of the examples are also available from the site. The site will soon include presentation slides for teaching, as well as other instructional material like examples and illustrative studies. The URL of the website is:

http://www.cse.iitk.ac.in/JaloteSEbook

I would like to express my gratitude to many people who helped me in preparing the case study. These include Kapil Narula, Ragesh Jaiswal, Vivek Pandey, Nilesh Lunawat, and Rajneesh Malviya. My special thanks to Vipindeep Vangala and Raghu Lingampally whose help in manuscript and Web site preparation allowed me to focus on the contents.

Pankaj Jalote

1

Introduction

Ask any student who has had some programming experience the following question: You are given a problem for which you have to build a software system that most students feel will be approximately 10,000 lines of (say C or Java) code. If you are working full time on it, how long will it take you to build this system?

The answer of students is generally 1 to 3 months. And, given the programming expertise of the students, there is a good chance that they will be able to build a system and demo it to the Professor within 2 months. With 2 months as the completion time, the productivity of the student will be 5,000 lines of code (LOC) per person-month.

Now let us take an alternative scenario—we act as clients and pose the same problem to a company that is in the business of developing software for clients. Though there is no "standard" productivity figure and it varies a lot, it is fair to say a productivity figure of 1,000 LOC per person-month is quite respectable (though it can be as low as 100 LOC per person-month for embedded systems). With this productivity, a team of professionals in a software organization will take 10 person-months to build this software system.

Why this difference in productivity in the two scenarios? Why is it that the same students who can produce software at a productivity of a few thousand LOC per month while in college end up producing only about a thousand LOC per month when working in a company? Why is it that students seem to be more productive in their student days than when they become professionals?

The answer, of course, is that two different things are being built in the

two scenarios. In the first, a *student system* is being built whose main purpose is to demo that it works. In the second scenario, a team of professionals in an organization is building the system for a client who is paying for it, and whose business may depend on proper working of the system. As should be evident, building the latter type of software is a different problem altogether. It is this problem in which software engineering is interested. The difference between the two types of software was recognized early and the term *software engineering* was coined at NATO sponsored conferences held in Europe in the 1960s to discuss the growing software crisis and the need to focus on software development.

In the rest of the chapter we further define our problem domain. Then we discuss some of the key factors that drive software engineering. This is followed by the basic approach followed by software engineering. In the rest of the book we discuss in more detail the various aspects of the software engineering approach.

1.1 The Problem Domain

In software engineering we are not dealing with programs that people build to illustrate something or for hobby (which we are referring to as student systems). Instead the problem domain is the software that solves some problem of some users where larger systems or businesses may depend on the software, and where problems in the software can lead to significant direct or indirect loss. We refer to this software as *industrial strength software.* Let us first discuss the key difference between the student software and the industrial strength software.

1.1.1 Industrial Strength Software

A student system is primarily meant for demonstration purposes; it is generally not used for solving any real problem of any organization. Consequently, nothing of significance or importance depends on proper functioning of the software. Because nothing of significance depends on the software, the presence of "bugs" (or defects or faults) is not a major concern. Hence the software is generally not designed with quality issues like portability, robustness, reliability, and usability in mind. Also, the student software system is generally used by the developer him- or herself, therefore the need for documentation is nonexistent, and again bugs are not critical issues as the user can fix them as and when they are found.

An *industrial strength software system*, on the other hand, is built to solve some problem of a client and is used by the clients organization for operating some part of business (we use the term "business" in a very broad sense—it may be to manage inventories, finances, monitor patients, air traffic control, etc.) In other words, important activities depend on the correct functioning of the system. And a malfunction of such a system can have huge impact in terms of financial or business loss, inconvenience to users, or loss of property and life. Consequently, the software system needs to be of high quality with respect to properties like dependability, reliability, user-friendliness, etc.

This requirement of high quality has many ramifications. First, it requires that the software be thoroughly tested before being used. The need for rigorous testing increases the cost considerably. In an industrial strength software project, 30% to 50% of the total effort may be spent in testing (while in a student software even 5% may be too high!)

Second, building high quality software requires that the development be broken into phases such that output of each phase is evaluated and reviewed so bugs can be removed. This desire to partition the overall problem into phases and identify defects early requires more documentation, standards, processes, etc. All these increase the effort required to build the software—hence the productivity of producing industrial strength software is generally much lower than for producing student software.

Industrial strength software also has other properties which do not exist in student software systems. Typically, for the same problem, the detailed requirements of what the software should do increase considerably. Besides quality requirements, there are requirements of backup and recovery, fault tolerance, following of standards, portability, etc. These generally have the effect of making the software system more complex and larger. The size of the industrial strength software system may be two times or more than the student system for the same problem.

Overall, if we assume one-fifth productivity, and an increase in size by a factor of two for the same problem, an industrial strength software system will take about 10 times as much effort to build as a student software system for the same problem. The rule of thumb Brooks gives also says that industrial strength software may cost about 10 times the student software[25]. The software industry is largely interested in developing industrial strength software, and the area of software engineering focuses on how to build such systems. In the rest of the book, when we use the term software, we mean industrial strength software.

IEEE defines *software* as the collection of computer programs, proce-