Software Engineering

Software engineering is as much about teamwork as it is about technology. This introductory textbook covers both. For courses featuring a team project, it offers tips and templates for aligning classroom concepts with the needs of the students' projects. Students will learn how software is developed in industry by adopting agile methods, discovering requirements, designing modular systems, selecting effective tests, and using metrics to track progress. The book also covers the why's behind the how-to's, to prepare students for advances in industry practices. The chapters explore ways of eliciting what users really want, how clean architecture divides and conquers the inherent complexity of software systems, how test coverage is essential for detecting the inevitable defects in code, and much more. Ravi Sethi provides real-life case studies and examples to demonstrate practical applications of the concepts. Online resources include sample project materials for students, and lecture slides for instructors.

Ravi Sethi is Laureate Professor of Computer Science at the University of Arizona, USA, and is an ACM fellow. He co-authored *Compilers: Principles, Techniques, and Tools*, popularly known as the "dragon" book, and launched Avaya Labs.

Software Engineering

Basic Principles and Best Practices

RAVI SETHI
University of Arizona

CAMBRIDGE
UNIVERSITY PRESS

CAMBRIDGE
UNIVERSITY PRESS

Shaftesbury Road, Cambridge CB2 8EA, United Kingdom

One Liberty Plaza, 20th Floor, New York, NY 10006, USA

477 Williamstown Road, Port Melbourne, VIC 3207, Australia

314–321, 3rd Floor, Plot 3, Splendor Forum, Jasola District Centre, New Delhi – 110025, India

103 Penang Road, #05–06/07, Visioncrest Commercial, Singapore 238467

Cambridge University Press is part of Cambridge University Press & Assessment,
a department of the University of Cambridge.

We share the University's mission to contribute to society through the pursuit of
education, learning and research at the highest international levels of excellence.

www.cambridge.org
Information on this title: www.cambridge.org/highereducation/isbn/9781316511947
DOI: 10.1017/9781009051811

First published 2023

Printed in the United Kingdom by TJ Books Limited, Padstow Cornwall 2023

A catalogue record for this publication is available from the British Library.

Library of Congress Cataloging-in-Publication Data
Names: Sethi, Ravi Mohan, 1947-author.
Title: Software engineering : basic principles and best practices /
Ravi Sethi, University of Arizona.
Description: Cambridge, United Kingdom ; New York, NY : Cambridge
University Press, 2023. | Includes bibliographical references and index.
Identifiers: LCCN 2022034132 (print) | LCCN 2022034133 (ebook) |
ISBN 9781316511947 (hardback) | ISBN 9781009051811 (epub)
Subjects: LCSH: Software engineering.
Classification: LCC QA76.758 .S45834 2023 (print) | LCC QA76.758 (ebook) |
DDC 005.1–dc23/eng/20220822
LC record available at https://lccn.loc.gov/2022034132
LC ebook record available at https://lccn.loc.gov/2022034133

ISBN 978-1-316-51194-7 Hardback

Additional resources for this publication at www.cambridge.org/sethi.

Brief Contents

Contents

Preface

The selection of content for this book was guided by the following question: What do software engineers really need to know about the subject to be productive today and relevant tomorrow? The discipline continues to evolve, driven by new applications, technologies, and development methods. There is every indication that the evolution will continue during an engineer's career. The IEEE-ACM software engineering curriculum guidelines stress continual learning:

Because so much of what is learned will change over a student's professional career and only a small fraction of what could be learned will be taught and learned at university, it is of paramount importance that students develop the habit of continually expanding their knowledge.[1]

This book therefore focuses on basic principles and best practices. The emphasis is not only on what works, but on why it works. The book includes real-world examples and case studies, where possible. Some classic examples are included for perspective.

Principles endure while practices evolve as the assumptions behind them are reexamined. The principles in the book relate to the intrinsic properties of software and human nature: software is complex, requirements change, defects are inevitable, teams need coordination. Assumptions about how to deal with these intrinsic properties have been tested over the years. Must testing follow coding? Not with test-driven development. The distinction between development and maintenance blurs with an evolving software code base. All assumptions have to be questioned to enable the pace of continuous deployment. What does not change is that design and architecture are the key to managing complexity, iterative agile methods accommodate requirements changes, validation and verification reduce defects, and a healthy balance of structure and flexibility motivates teams and improves performance.

Content Organization and Coverage

This book is intended for a junior- or senior-level introductory course in software engineering. Students are expected to have enough programming maturity to engage in a team project. They are not expected to have any prior team experience.

The ACM-IEEE guidelines strongly recommend the inclusion of a significant project in a software engineering course. First, systematic engineering methods are intended for problems of complexity and scale. With a significant project, students get to experience the benefits of engineering concepts and methods. Second, users and teams bring a human dimension to the discipline. Working with a real customer on a project suitable for a team of, say, four provides students with a team experience. Appendix A addresses the challenge of organizing a course with dual tracks for concepts and a project. See also the brief comments in the following paragraphs.

The chapters in this book can be grouped as follows: getting started, what to build, design and architecture, software quality, and metrics.

Getting Started: Chapters 1–2 Chapter 1 introduces key topics that are explored in the rest of the book: requirements, software architecture, and testing. The chapter also has a section on social responsibility and professional conduct.

Chapter 2 deals with processes, which orchestrate team activities. A team's culture and values guide activities that are not covered by the rules of a process. Process models tend to focus on specific activities, leaving the rest to the team: Scrum focuses on planning and review events, XP on development practices, V processes on testing, and the Spiral Framework on risk reduction. The chapter discusses how a team can combine best practices from these process models for its project.

What to Build? Chapters 3–5 Requirements development is iterative, with both agile and plan-driven methods. The difference is that agile methods favor working software to validate what to build, whereas plan-driven methods validate a specification document. Chapter 3 deals with elicitation (discovery) of the wants and goals that users communicate through their words, actions, and emotions. What users say can differ from what they do and feel. To help clarify user goals, the chapter introduces three classes of questions, aimed at identifying user motivations, solution options, and goal quantification. User requirements can be recorded as user stories, system features, or user-experience scenarios. Goal refinement techniques, covered with requirements, also apply to security (attack trees) and measurement (metrics).

The requirements prioritization techniques in Chapter 4 include MoSCoW (must-should-could-won't), value-cost balancing, value-cost-risk assessment, and Kano analysis. Kano analysis is based not only on what satisfies customers, but on what dissatisfies them. The chapter also includes estimation techniques based on story points for agile methods and on Cocomo formal models for plan-driven methods. Anchoring, which can bias estimates, is a phenomenon to be avoided during both individual and group estimation.

Chapter 5 covers use cases. Use cases can be lightweight if they are developed incrementally, starting with user goals, then adding basic flows, and finally alternative flows as needed.

Design and Architecture: Chapters 6–7 Architecture is a subset of design, so the following comments carry over to design. Software architecture, Chapter 6, is key to managing the complexity of software. A modular system is built up from units such as classes, so the chapter introduces UML (Unified Modeling Language) class diagrams and includes guidelines for designing modular systems. For system architecture, the chapter introduces views and how to describe a system in terms of key view(s).

A pattern outlines a solution to a problem that occurs over and over again. Chapter 7 covers the following architectural patterns: layering, shared data, observer, publish-subscribe, model-view-controller, client-server, and broker. Client–server architectures enable new software to be deployed to a production system: a load balancer directs some of the incoming traffic to the new software on a trial basis, until the new software is ready to go into production. The ability to add new software to a production system is needed for continuous deployment.

Software Quality: Chapters 8–10 The combination of reviews (architecture and code), static analysis, and testing is much more effective for defect detection than any of these techniques by themselves. Chapter 8 discusses reviews and static analysis. The focus of the chapter is on static or compile-time techniques.

Chapter 9 is on testing, which is done by running code on specific test inputs. A test set is considered good enough if it meets the desired coverage criteria. For code coverage, the chapter includes statement, branch (decision), and MC/DC coverage. For input-domain coverage, the chapter includes equivalence partitioning and combinatorial testing.

The quality theme continues into Chapter 10, which applies metrics and measurement to the goal of quality assessment and improvement. The chapter introduces six forms of quality: functional, process, product, operations (ops), aesthetics, and customer satisfaction. Ops quality refers to quality after a system is installed at a customer site.

Metrics: Chapter 10 An alternative long title for this chapter is "the design and use of metrics and measurement, with applications to software quality." The chapter has roughly three parts. The first part introduces the measurement process, the design of useful metrics, and the graphical display of data sets. The second part deals with metrics for product and ops quality. The third part introduces statistical techniques. Boxplots, histograms, variance, and standard deviation summarize the dispersion of values in data sets. The last two sections on confidence intervals and simple linear regression are mathematical.

A Team Project: Appendix A The main challenge in a course with a concepts track and a project track is that the two tracks have somewhat different objectives; for example, the concepts track typically covers multiple process models, while a project embraces a single process. In general, each of the two tracks has its own pace: the concepts track takes many weeks to cover the what, why, and how of processes, requirements, design, and testing, whereas some knowledge of all of these topics is needed early, during the first iteration of a project.

The appendix discusses a hybrid approach, which aligns the tracks initially, while teams are formed and projects are launched. Once the students have enough to start their projects, the coupling between the tracks can be loosened so each track can proceed at its own pace.

Acknowledgments

My trial-by-software came when Harry Huskey handed me a full box of cards with the assembly code for a compiler for HH-1, his Algol-like language. The code was the only description for either the language or the compiler. Harry then left for a long trip, after giving me free rein to see what I could do. Hello, summer job! The job was at the end of my first year as an undergraduate at IIT Kanpur. At the time, the sum total of my experience was a few short Fortran and assembly programs for the IBM 1620. I spent a formative summer tracing through the code and doing peephole optimization. In retrospect, it would have helped if I had clarified Harry's requirements, tested after every change, asked for a view of the system architecture, and so on.

Fast forward a dozen years to Bell Labs in New Jersey, where I had the good fortune to join Doug McIlroy's department in the Computer Science Research Center. In the Center, Unix was about to be ported from one machine to another for the first time. As a user, I came to appreciate the Unix tools culture, quick iterations, and continual refinement of tools through user feedback. Steve Johnson once lamented "the inability of users to see things" his way and added that "invariably, they were right!"

I also came to appreciate the challenges faced by the Switching business unit in building 99.999 percent reliable telephone switches: each phone call was handled by a concurrent process at each end. Their existing approach had a cast of thousands using variants of waterfall processes. They managed the risks of waterfall methods by freezing requirements and by rigorous continuous validation and verification.

In 2000, Avaya was spun off as a separate company and I went with it to build up Avaya Labs Research. Half of the research staff came from Bell Labs, David Weiss among them. David built up the software technology research department, together with a small group called the Avaya Resource Center (ARC), which served as a bridge to the R&D community in the business units. The mission for the department and the ARC was to "Improve the state of software in Avaya and know it." After David retired, I got a chance to work directly with Randy Hackbarth and John Palframan from the ARC and with Audris Mockus and Jenny Li from the research department. These were the people behind the quality improvement effort described in Section 10.5. I am grateful to them for their briefings and insights about software.

When I joined the University of Arizona in 2014, David generously shared the materials from his software engineering course at Iowa State University. At Arizona, I want to thank Todd Proebsting and David Lowenthal for the opportunity to teach the senior and graduate-level software engineering courses.

For this book, I would like to single out Jon Bentley, Audris Mockus, and Joann Ordille for their insights, encouragement, and comments. Thanks also to the many students in the software engineering courses. The Cambridge University Press team has been most thorough. My thanks to the editors, Emily Watton and Lauren Cowles. Emily's ongoing suggestions have greatly improved the book. Most of all, I want to thank Dianne and Alexandra. Alexandra took over the figures when I could no longer do them.