



GLOBAL
EDITION



Software Engineering

TENTH EDITION

Ian Sommerville

ALWAYS LEARNING

PEARSON



SOFTWARE ENGINEERING

Tenth Edition

Ian Sommerville

PEARSON

Boston Columbus Indianapolis New York San Francisco Hoboken
Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montreal Toronto
Delhi Mexico City São Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo

Editorial Director: Marcia Horton
Editor in Chief: Michael Hirsch
Acquisitions Editor: Matt Goldstein
Editorial Assistant: Chelsea Bell
Assistant Acquisitions Editor, Global Edition: Murchana Borthakur
Associate Project Editor, Global Edition: Binita Roy
Managing Editor: Jeff Holcomb
Senior Production Project Manager: Marilyn Lloyd
Director of Marketing: Margaret Waples

Marketing Coordinator: Kathryn Ferranti
Senior Manufacturing Buyer: Carol Melville
Senior Manufacturing Controller, Production, Global Edition: Trudy Kimber
Text Designer: Susan Raymond
Cover Art Designer: Lumina Datamatics
Cover Image: © Andrey Bayda/Shutterstock
Interior Chapter Opener: © graficart.net/Alamy
Full-Service Project Management: Rashmi Tickyani, Aptara®, Inc.
Composition and Illustrations: Aptara®, Inc.

Pearson Education Limited
Edinburgh Gate
Harlow
Essex CM20 2JE
England

and Associated Companies throughout the world

Visit us on the World Wide Web at:
www.pearsonglobaleditions.com

© Pearson Education Limited 2016

The rights of Ian Sommerville to be identified as the author of this work have been asserted by him in accordance with the Copyright, Designs and Patents Act 1988.

Authorized adaptation from the United States edition, entitled Software Engineering, 10th edition, ISBN 978-0-13-394303-0, by Ian Sommerville, published by Pearson Education © 2016.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without either the prior written permission of the publisher or a license permitting restricted copying in the United Kingdom issued by the Copyright Licensing Agency Ltd, Saffron House, 6–10 Kirby Street, London EC1N 8TS.

All trademarks used herein are the property of their respective owners. The use of any trademark in this text does not vest in the author or publisher any trademark ownership rights in such trademarks, nor does the use of such trademarks imply any affiliation with or endorsement of this book by such owners.

ISBN 10: 1-292-09613-6
ISBN 13: 978-1-292-09613-1

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library

10 9 8 7 6 5 4 3 2 1

Typeset in 9 New Aster LT Std by Aptara®, Inc.

Printed and bound by Courier Westford in the United States of America.



PREFACE

Progress in software engineering over the last 50 years has been astonishing. Our societies could not function without large professional software systems. National utilities and infrastructure—energy, communications and transport—all rely on complex and mostly reliable computer systems. Software has allowed us to explore space and to create the World Wide Web—the most significant information system in the history of mankind. Smartphones and tablets are ubiquitous and an entire ‘apps industry’ developing software for these devices has emerged in the past few years.

Humanity is now facing a demanding set of challenges—climate change and extreme weather, declining natural resources, an increasing world population to be fed and housed, international terrorism, and the need to help elderly people lead satisfying and fulfilled lives. We need new technologies to help us address these challenges and, for sure, software will have a central role in these technologies. Software engineering is, therefore, critically important for our future on this planet. We have to continue to educate software engineers and develop the discipline so that we meet the demand for more software and create the increasingly complex future systems that we need.

Of course, there are still problems with software projects. Systems are still sometimes delivered late and cost more than expected. We are creating increasingly complex software systems of systems and we should not be surprised that we encounter difficulties along the way. However, we should not let these problems conceal the real successes in software engineering and the impressive software engineering methods and technologies that have been developed.

This book, in different editions, has now been around for over 30 years and this edition is based around the essential principles that were established in the first edition:

1. I write about software engineering as it is practiced in industry, without taking an evangelical position on particular approaches such as agile development or formal methods. In reality, industry mixes techniques such as agile and plan-based development and this is reflected in the book.

2. I write about what I know and understand. I have had many suggestions for additional topics that might be covered in more detail such as open source development, the use of the UML and mobile software engineering. But I don't really know enough about these areas. My own work has been in system dependability and in systems engineering and this is reflected in my selection of advanced topics for the book.

I believe that the key issues for modern software engineering are managing complexity, integrating agility with other methods and ensuring that our systems are secure and resilient. These issues have been the driver for the changes and additions in this new edition of my book.

Changes from the 9th edition

In summary, the major updates and additions in this book from the 9th edition are:

- I have extensively updated the chapter on agile software engineering, with new material on Scrum. I have updated other chapters as required to reflect the increasing use of agile methods of software engineering.
- I have added new chapters on resilience engineering, systems engineering, and systems of systems.
- I have completely reorganized three chapters covering reliability, safety, and security.
- I have added new material on RESTful services to the chapter covering service-oriented software engineering.
- I have revised and updated the chapter on configuration management with new material on distributed version control systems.
- I have moved chapters on aspect-oriented software engineering and process improvement from the print version of the book to the web site.
- New supplementary material has been added to the web site, including a set of supporting videos. I have explained key topics on video and recommended related YouTube videos.

The 4-part structure of the book, introduced in earlier editions, has been retained but I have made significant changes in each part of the book.

1. In Part 1, Introduction to software engineering, I have completely rewritten Chapter 3 (agile methods) and updated this to reflect the increasing use of Scrum. A new case study on a digital learning environment has been added to Chapter 1 and is used in a number of chapters. Legacy systems are covered in more detail in Chapter 9. Minor changes and updates have been made to all other chapters.

2. Part 2, which covers dependable systems, has been revised and restructured. Rather than an activity-oriented approach where information on safety, security and reliability is spread over several chapters, I have reorganized this so that each topic has a chapter in its own right. This makes it easier to cover a single topic, such as security, as part of a more general course. I have added a completely new chapter on resilience engineering which covers cybersecurity, organizational resilience, and resilient systems design.
3. In Part 3, I have added new chapters on systems engineering and systems of systems and have extensively revised the material on service-oriented systems engineering to reflect the increasing use of RESTful services. The chapter on aspect-oriented software engineering has been deleted from the print version but remains available as a web chapter.
4. In Part 4, I have updated the material on configuration management to reflect the increasing use of distributed version control tools such as Git. The chapter on process improvement has been deleted from the print version but remains available as a web chapter.

An important change in the supplementary material for the book is the addition of video recommendations in all chapters. I have made over 40 videos on a range of topics that are available on my YouTube channel and linked from the book's web pages. In cases where I have not made videos, I have recommended YouTube videos that may be useful.

I explain the rationale behind the changes that I've made in this short video:

<http://software-engineering-book/videos/10th-edition-changes>

Readership

The book is primarily aimed at university and college students taking introductory and advanced courses in software and systems engineering. I assume that readers understand the basics of programming and fundamental data structures.

Software engineers in industry may find the book useful as general reading and to update their knowledge on topics such as software reuse, architectural design, dependability and security and systems engineering.

Using the book in software engineering courses

I have designed the book so that it can be used in three different types of software engineering course:

1. *General introductory courses in software engineering.* The first part of the book has been designed to support a 1-semester course in introductory software engineering. There are 9 chapters that cover fundamental topics in software engineering.

If your course has a practical component, management chapters in Part 4 may be substituted for some of these.

2. *Introductory or intermediate courses on specific software engineering topics.* You can create a range of more advanced courses using the chapters in parts 2–4. For example, I have taught a course in critical systems using the chapters in Part 2 plus chapters on systems engineering and quality management. In a course covering software-intensive systems engineering, I used chapters on systems engineering, requirements engineering, systems of systems, distributed software engineering, embedded software, project management and project planning.
3. *More advanced courses in specific software engineering topics.* In this case, the chapters in the book form a foundation for the course. These are then supplemented with further reading that explores the topic in more detail. For example, a course on software reuse could be based around Chapters 15–18.

Instructors may access additional teaching support material from Pearson’s website. Some of this is password-protected and instructors using the book for teaching can obtain a password by registering at the Pearson website. The material available includes:

- Model answers to selected end of chapter exercises.
- Quiz questions and answers for each chapter.

You can access this material at:

www.pearsonglobaleditions.com/Sommerville

Book website

This book has been designed as a hybrid print/web text in which core information in the printed edition is linked to supplementary material on the web. Several chapters include specially written ‘web sections’ that add to the information in that chapter. There are also six ‘web chapters’ on topics that I have not covered in the print version of the book.

You can download a wide range of supporting material from the book’s website (software-engineering-book.com) including:

- A set of videos where I cover a range of software engineering topics. I also recommend other YouTube videos that can support learning.
- An instructor’s guide that gives advice on how to use the book in teaching different courses.
- Further information on the book’s case studies (insulin pump, mental health care system, wilderness weather system, digital learning system), as well other case studies, such as the failure of the Ariane 5 launcher.

- Six web chapters covering process improvement, formal methods, interaction design, application architectures, documentation and aspect-oriented development.
- Web sections that add to the content presented in each chapter. These web sections are linked from breakout boxes in each chapter.
- PowerPoint presentations for all of the chapters in the book and additional PowerPoint presentations covering a range of systems engineering topics are available at pearsonglobaleditions.com/Sommerville.

In response to requests from users of the book, I have published a complete requirements specification for one of the system case studies on the book's web site. It is difficult for students to get access to such documents and so understand their structure and complexity. To avoid confidentiality issues, I have re-engineered the requirements document from a real system so there are no restrictions on its use.

Contact details

Website: software-engineering-book.com

Email: name: [software.engineering.book](mailto:software.engineering.book@gmail.com); domain: [gmail.com](mailto:software.engineering.book@gmail.com)

Blog: iansommerville.com/systems-software-and-technology

YouTube: youtube.com/user/SoftwareEngBook

Facebook: facebook.com/sommerville.software.engineering

Twitter: [@SoftwareEngBook](https://twitter.com/SoftwareEngBook) or [@iansommerville](https://twitter.com/iansommerville) (for more general tweets)

Follow me on Twitter or Facebook to get updates on new material and comments on software and systems engineering.

Acknowledgements

A large number of people have contributed over the years to the evolution of this book and I'd like to thank everyone (reviewers, students and book users) who have commented on previous editions and made constructive suggestions for change. I'd particularly like to thank my family, Anne, Ali, and Jane, for their love, help and support while I was working on this book (and all of the previous editions).

*Ian Sommerville,
September 2014*

Contents at a glance

	Preface	3
Part 1	Introduction to Software Engineering	15
	Chapter 1 Introduction	17
	Chapter 2 Software processes	43
	Chapter 3 Agile software development	72
	Chapter 4 Requirements engineering	101
	Chapter 5 System modeling	138
	Chapter 6 Architectural design	167
	Chapter 7 Design and implementation	196
	Chapter 8 Software testing	226
	Chapter 9 Software evolution	255
Part 2	System Dependability and Security	283
	Chapter 10 Dependable systems	285
	Chapter 11 Reliability engineering	306
	Chapter 12 Safety engineering	339
	Chapter 13 Security engineering	373
	Chapter 14 Resilience engineering	408
Part 3	Advanced Software Engineering	435
	Chapter 15 Software reuse	437
	Chapter 16 Component-based software engineering	464
	Chapter 17 Distributed software engineering	490
	Chapter 18 Service-oriented software engineering	520
	Chapter 19 Systems engineering	551
	Chapter 20 Systems of systems	580
	Chapter 21 Real-time software engineering	610
Part 4	Software Management	639
	Chapter 22 Project management	641
	Chapter 23 Project planning	667
	Chapter 24 Quality management	700
	Chapter 25 Configuration management	730
	Glossary	757
	Subject index	777
	Author index	803

Pearson wishes to thank and acknowledge the following people for their work on the Global Edition:

Contributor

Sherif G. Aly, The American University in Cairo

Muthuraj M., Android developer

Reviewers

Mohit P. Tahiliani, National Institute of Technology Karnataka, Surathkal

Chitra Dhawale, P. R. Patil Group of Educational Institutes, Amravati

Sanjeevni Shantaiya, Disha Institute of Management & Technology



CONTENTS

Preface	3
Part 1 Introduction to Software Engineering	15
Chapter 1 Introduction	17
1.1 Professional software development	19
1.2 Software engineering ethics	28
1.3 Case studies	31
Chapter 2 Software processes	43
2.1 Software process models	45
2.2 Process activities	54
2.3 Coping with change	61
2.4 Process improvement	65
Chapter 3 Agile software development	72
3.1 Agile methods	75
3.2 Agile development techniques	77
3.3 Agile project management	84
3.4 Scaling agile methods	88

Chapter 4	Requirements engineering	101
4.1	Functional and non-functional requirements	105
4.2	Requirements engineering processes	111
4.3	Requirements elicitation	112
4.4	Requirements specification	120
4.5	Requirements validation	129
4.6	Requirements change	130
Chapter 5	System modeling	138
5.1	Context models	141
5.2	Interaction models	144
5.3	Structural models	149
5.4	Behavioral models	154
5.5	Model-driven architecture	159
Chapter 6	Architectural design	167
6.1	Architectural design decisions	171
6.2	Architectural views	173
6.3	Architectural patterns	175
6.4	Application architectures	184
Chapter 7	Design and implementation	196
7.1	Object-oriented design using the UML	198
7.2	Design patterns	209
7.3	Implementation issues	212
7.4	Open-source development	219
Chapter 8	Software testing	226
8.1	Development testing	231
8.2	Test-driven development	242

8.3	Release testing	245
8.4	User testing	249
Chapter 9	Software evolution	255
9.1	Evolution processes	258
9.2	Legacy systems	261
9.3	Software maintenance	270
Part 2	System Dependability and Security	283
Chapter 10	Dependable systems	285
10.1	Dependability properties	288
10.2	Sociotechnical systems	291
10.3	Redundancy and diversity	295
10.4	Dependable processes	297
10.5	Formal methods and dependability	299
Chapter 11	Reliability engineering	306
11.1	Availability and reliability	309
11.2	Reliability requirements	312
11.3	Fault-tolerant architectures	318
11.4	Programming for reliability	325
11.5	Reliability measurement	331
Chapter 12	Safety engineering	339
12.1	Safety-critical systems	341
12.2	Safety requirements	344
12.3	Safety engineering processes	352
12.4	Safety cases	361

Chapter 13	Security engineering	373
13.1	Security and dependability	376
13.2	Security and organizations	380
13.3	Security requirements	382
13.4	Secure systems design	388
13.5	Security testing and assurance	402
Chapter 14	Resilience engineering	408
14.1	Cybersecurity	412
14.2	Sociotechnical resilience	416
14.3	Resilient systems design	424
Part 3	Advanced Software Engineering	435
<hr/>		
Chapter 15	Software reuse	437
15.1	The reuse landscape	440
15.2	Application frameworks	443
15.3	Software product lines	446
15.4	Application system reuse	453
Chapter 16	Component-based software engineering	464
16.1	Components and component models	467
16.2	CBSE processes	473
16.3	Component composition	480
Chapter 17	Distributed software engineering	490
17.1	Distributed systems	492
17.2	Client–server computing	499

17.3 Architectural patterns for distributed systems	501
17.4 Software as a service	512
Chapter 18 Service-oriented software engineering	520
18.1 Service-oriented architecture	524
18.2 RESTful services	529
18.3 Service engineering	533
18.4 Service composition	541
Chapter 19 Systems engineering	551
19.1 Sociotechnical systems	556
19.2 Conceptual design	563
19.3 System procurement	566
19.4 System development	570
19.5 System operation and evolution	574
Chapter 20 Systems of systems	580
20.1 System complexity	584
20.2 Systems of systems classification	587
20.3 Reductionism and complex systems	590
20.4 Systems of systems engineering	593
20.5 Systems of systems architecture	599
Chapter 21 Real-time software engineering	610
21.1 Embedded system design	613
21.2 Architectural patterns for real-time software	620
21.3 Timing analysis	626
21.4 Real-time operating systems	631

Part 4	Software Management	639
Chapter 22	Project management	641
	22.1 Risk management	644
	22.2 Managing people	652
	22.3 Teamwork	656
Chapter 23	Project planning	667
	23.1 Software pricing	670
	23.2 Plan-driven development	672
	23.3 Project scheduling	675
	23.4 Agile planning	680
	23.5 Estimation techniques	682
	23.6 COCOMO cost modeling	686
Chapter 24	Quality management	700
	24.1 Software quality	703
	24.2 Software standards	706
	24.3 Reviews and inspections	710
	24.4 Quality management and agile development	714
	24.5 Software measurement	716
Chapter 25	Configuration management	730
	25.1 Version management	735
	25.2 System building	740
	25.3 Change management	745
	25.4 Release management	750
	Glossary	757
	Subject index	777
	Author index	803



PART

1

Introduction to Software Engineering

My aim in this part of the book is to provide a general introduction to software engineering. The chapters in this part have been designed to support a one-semester first course in software engineering. I introduce important concepts such as software processes and agile methods, and describe essential software development activities, from requirements specification through to system evolution.

Chapter 1 is a general introduction that introduces professional software engineering and defines some software engineering concepts. I have also included a brief discussion of ethical issues in software engineering. It is important for software engineers to think about the wider implications of their work. This chapter also introduces four case studies that I use in the book. These are an information system for managing records of patients undergoing treatment for mental health problems (Mentcare), a control system for a portable insulin pump, an embedded system for a wilderness weather station and a digital learning environment (iLearn).

Chapters 2 and 3 cover software engineering processes and agile development. In Chapter 2, I introduce software process models, such as the waterfall model, and I discuss the basic activities that are part of these processes. Chapter 3 supplements this with a discussion of agile development methods for software engineering. This chapter had been

extensively changed from previous editions with a focus on agile development using Scrum and a discussion of agile practices such as stories for requirements definition and test-driven development.

The remaining chapters in this part are extended descriptions of the software process activities that are introduced in Chapter 2. Chapter 4 covers the critically important topic of requirements engineering, where the requirements for what a system should do are defined. Chapter 5 explains system modeling using the UML, where I focus on the use of use case diagrams, class diagrams, sequence diagrams and state diagrams for modeling a software system. In Chapter 6, I discuss the importance of software architecture and the use of architectural patterns in software design.

Chapter 7 introduces object oriented design and the use of design patterns. I also introduce important implementation issues here—reuse, configuration management and host-target development and discuss open source development. Chapter 8 focuses on software testing from unit testing during system development to the testing of software releases. I also discuss the use of test-driven development—an approach pioneered in agile methods but which has wide applicability. Finally, Chapter 9 presents an overview of software evolution issues. I cover evolution processes, software maintenance and legacy system management.



1

Introduction

Objectives

The objectives of this chapter are to introduce software engineering and to provide a framework for understanding the rest of the book. When you have read this chapter, you will:

- understand what software engineering is and why it is important;
- understand that the development of different types of software system may require different software engineering techniques;
- understand ethical and professional issues that are important for software engineers;
- have been introduced to four systems, of different types, which are used as examples throughout the book.

Contents

- 1.1 Professional software development
- 1.2 Software engineering ethics
- 1.3 Case studies

Software engineering is essential for the functioning of government, society, and national and international businesses and institutions. We can't run the modern world without software. National infrastructures and utilities are controlled by computer-based systems, and most electrical products include a computer and controlling software. Industrial manufacturing and distribution is completely computerized, as is the financial system. Entertainment, including the music industry, computer games, and film and television, is software-intensive. More than 75% of the world's population have a software-controlled mobile phone, and, by 2016, almost all of these will be Internet-enabled.

Software systems are abstract and intangible. They are not constrained by the properties of materials, nor are they governed by physical laws or by manufacturing processes. This simplifies software engineering, as there are no natural limits to the potential of software. However, because of the lack of physical constraints, software systems can quickly become extremely complex, difficult to understand, and expensive to change.

There are many different types of software system, ranging from simple embedded systems to complex, worldwide information systems. There are no universal notations, methods, or techniques for software engineering because different types of software require different approaches. Developing an organizational information system is completely different from developing a controller for a scientific instrument. Neither of these systems has much in common with a graphics-intensive computer game. All of these applications need software engineering; they do not all need the same software engineering methods and techniques.

There are still many reports of software projects going wrong and of “software failures.” Software engineering is criticized as inadequate for modern software development. However, in my opinion, many of these so-called software failures are a consequence of two factors:

1. *Increasing system complexity* As new software engineering techniques help us to build larger, more complex systems, the demands change. Systems have to be built and delivered more quickly; larger, even more complex systems are required; and systems have to have new capabilities that were previously thought to be impossible. New software engineering techniques have to be developed to meet new the challenges of delivering more complex software.
2. *Failure to use software engineering methods* It is fairly easy to write computer programs without using software engineering methods and techniques. Many companies have drifted into software development as their products and services have evolved. They do not use software engineering methods in their everyday work. Consequently, their software is often more expensive and less reliable than it should be. We need better software engineering education and training to address this problem.

Software engineers can be rightly proud of their achievements. Of course, we still have problems developing complex software, but without software engineering we would not have explored space and we would not have the Internet or modern telecommunications. All forms of travel would be more dangerous and expensive. Challenges for humanity in the 21st century are climate change, fewer natural



History of software engineering

The notion of software engineering was first proposed in 1968 at a conference held to discuss what was then called the software crisis (Naur and Randell 1969). It became clear that individual approaches to program development did not scale up to large and complex software systems. These were unreliable, cost more than expected, and were delivered late.

Throughout the 1970s and 1980s, a variety of new software engineering techniques and methods were developed, such as structured programming, information hiding, and object-oriented development. Tools and standard notations were developed which are the basis of today's software engineering.

<http://software-engineering-book.com/web/history/>

resources, changing demographics, and an expanding world population. We will rely on software engineering to develop the systems that we need to cope with these issues.

1.1 Professional software development

Lots of people write programs. People in business write spreadsheet programs to simplify their jobs; scientists and engineers write programs to process their experimental data; hobbyists write programs for their own interest and enjoyment. However, most software development is a professional activity in which software is developed for business purposes, for inclusion in other devices, or as software products such as information systems and computer-aided design systems. The key distinctions are that professional software is intended for use by someone apart from its developer and that teams rather than individuals usually develop the software. It is maintained and changed throughout its life.

Software engineering is intended to support professional software development rather than individual programming. It includes techniques that support program specification, design, and evolution, none of which are normally relevant for personal software development. To help you to get a broad view of software engineering, I have summarized frequently asked questions about the subject in Figure 1.1.

Many people think that software is simply another word for computer programs. However, when we are talking about software engineering, software is not just the programs themselves but also all associated documentation, libraries, support websites, and configuration data that are needed to make these programs useful. A professionally developed software system is often more than a single program. A system may consist of several separate programs and configuration files that are used to set up these programs. It may include system documentation, which describes the structure of the system, user documentation, which explains how to use the system, and websites for users to download recent product information.

This is one of the important differences between professional and amateur software development. If you are writing a program for yourself, no one else will use it

Question	Answer
What is software?	Computer programs and associated documentation. Software products may be developed for a particular customer or may be developed for a general market.
What are the attributes of good software?	Good software should deliver the required functionality and performance to the user and should be maintainable, dependable and usable.
What is software engineering?	Software engineering is an engineering discipline that is concerned with all aspects of software production from initial conception to operation and maintenance.
What are the fundamental software engineering activities?	Software specification, software development, software validation and software evolution.
What is the difference between software engineering and computer science?	Computer science focuses on theory and fundamentals; software engineering is concerned with the practicalities of developing and delivering useful software.
What is the difference between software engineering and system engineering?	System engineering is concerned with all aspects of computer-based systems development including hardware, software and process engineering. Software engineering is part of this more general process.
What are the key challenges facing software engineering?	Coping with increasing diversity, demands for reduced delivery times and developing trustworthy software.
What are the costs of software engineering?	Roughly 60% of software costs are development costs, 40% are testing costs. For custom software, evolution costs often exceed development costs.
What are the best software engineering techniques and methods?	While all software projects have to be professionally managed and developed, different techniques are appropriate for different types of system. For example, games should always be developed using a series of prototypes whereas safety critical control systems require a complete and analyzable specification to be developed. There are no methods and techniques that are good for everything.
What differences has the Internet made to software engineering?	Not only has the Internet led to the development of massive, highly distributed, service-based systems, it has also supported the creation of an “app” industry for mobile devices which has changed the economics of software.

Figure 1.1 Frequently asked questions about software engineering

and you don’t have to worry about writing program guides, documenting the program design, and so on. However, if you are writing software that other people will use and other engineers will change, then you usually have to provide additional information as well as the code of the program.

Software engineers are concerned with developing software products, that is, software that can be sold to a customer. There are two kinds of software product:

1. *Generic products* These are stand-alone systems that are produced by a development organization and sold on the open market to any customer who is able to buy them. Examples of this type of product include apps for mobile devices, software for PCs such as databases, word processors, drawing packages, and project management tools. This kind of software also includes “vertical”

applications designed for a specific market such as library information systems, accounting systems, or systems for maintaining dental records.

2. *Customized (or bespoke) software* These are systems that are commissioned by and developed for a particular customer. A software contractor designs and implements the software especially for that customer. Examples of this type of software include control systems for electronic devices, systems written to support a particular business process, and air traffic control systems.

The critical distinction between these types of software is that, in generic products, the organization that develops the software controls the software specification. This means that if they run into development problems, they can rethink what is to be developed. For custom products, the specification is developed and controlled by the organization that is buying the software. The software developers must work to that specification.

However, the distinction between these system product types is becoming increasingly blurred. More and more systems are now being built with a generic product as a base, which is then adapted to suit the requirements of a customer. Enterprise Resource Planning (ERP) systems, such as systems from SAP and Oracle, are the best examples of this approach. Here, a large and complex system is adapted for a company by incorporating information about business rules and processes, reports required, and so on.

When we talk about the quality of professional software, we have to consider that the software is used and changed by people apart from its developers. Quality is therefore not just concerned with what the software does. Rather, it has to include the software's behavior while it is executing and the structure and organization of the system programs and associated documentation. This is reflected in the software's quality or non-functional attributes. Examples of these attributes are the software's response time to a user query and the understandability of the program code.

The specific set of attributes that you might expect from a software system obviously depends on its application. Therefore, an aircraft control system must be safe, an interactive game must be responsive, a telephone switching system must be reliable, and so on. These can be generalized into the set of attributes shown in Figure 1.2, which I think are the essential characteristics of a professional software system.

1.1.1 Software engineering

Software engineering is an engineering discipline that is concerned with all aspects of software production from the early stages of system specification through to maintaining the system after it has gone into use. In this definition, there are two key phrases:

1. *Engineering discipline* Engineers make things work. They apply theories, methods, and tools where these are appropriate. However, they use them selectively

Product characteristic	Description
Acceptability	Software must be acceptable to the type of users for which it is designed. This means that it must be understandable, usable, and compatible with other systems that they use.
Dependability and security	Software dependability includes a range of characteristics including reliability, security, and safety. Dependable software should not cause physical or economic damage in the event of system failure. Software has to be secure so that malicious users cannot access or damage the system.
Efficiency	Software should not make wasteful use of system resources such as memory and processor cycles. Efficiency therefore includes responsiveness, processing time, resource utilization, etc.
Maintainability	Software should be written in such a way that it can evolve to meet the changing needs of customers. This is a critical attribute because software change is an inevitable requirement of a changing business environment.

Figure 1.2 Essential attributes of good software

and always try to discover solutions to problems even when there are no applicable theories and methods. Engineers also recognize that they must work within organizational and financial constraints, and they must look for solutions within these constraints.

2. *All aspects of software production* Software engineering is not just concerned with the technical processes of software development. It also includes activities such as software project management and the development of tools, methods, and theories to support software development.

Engineering is about getting results of the required quality within schedule and budget. This often involves making compromises—engineers cannot be perfectionists. People writing programs for themselves, however, can spend as much time as they wish on the program development.

In general, software engineers adopt a systematic and organized approach to their work, as this is often the most effective way to produce high-quality software. However, engineering is all about selecting the most appropriate method for a set of circumstances, so a more creative, less formal approach to development may be the right one for some kinds of software. A more flexible software process that accommodates rapid change is particularly appropriate for the development of interactive web-based systems and mobile apps, which require a blend of software and graphical design skills.

Software engineering is important for two reasons:

1. More and more, individuals and society rely on advanced software systems. We need to be able to produce reliable and trustworthy systems economically and quickly.
2. It is usually cheaper, in the long run, to use software engineering methods and techniques for professional software systems rather than just write programs as

a personal programming project. Failure to use software engineering method leads to higher costs for testing, quality assurance, and long-term maintenance.

The systematic approach that is used in software engineering is sometimes called a software process. A software process is a sequence of activities that leads to the production of a software product. Four fundamental activities are common to all software processes.

1. Software specification, where customers and engineers define the software that is to be produced and the constraints on its operation.
2. Software development, where the software is designed and programmed.
3. Software validation, where the software is checked to ensure that it is what the customer requires.
4. Software evolution, where the software is modified to reflect changing customer and market requirements.

Different types of systems need different development processes, as I explain in Chapter 2. For example, real-time software in an aircraft has to be completely specified before development begins. In e-commerce systems, the specification and the program are usually developed together. Consequently, these generic activities may be organized in different ways and described at different levels of detail, depending on the type of software being developed.

Software engineering is related to both computer science and systems engineering.

1. Computer science is concerned with the theories and methods that underlie computers and software systems, whereas software engineering is concerned with the practical problems of producing software. Some knowledge of computer science is essential for software engineers in the same way that some knowledge of physics is essential for electrical engineers. Computer science theory, however, is often most applicable to relatively small programs. Elegant theories of computer science are rarely relevant to large, complex problems that require a software solution.
2. System engineering is concerned with all aspects of the development and evolution of complex systems where software plays a major role. System engineering is therefore concerned with hardware development, policy and process design, and system deployment, as well as software engineering. System engineers are involved in specifying the system, defining its overall architecture, and then integrating the different parts to create the finished system.

As I discuss in the next section, there are many different types of software. There are no universal software engineering methods or techniques that may be used. However, there are four related issues that affect many different types of software:

1. *Heterogeneity* Increasingly, systems are required to operate as distributed systems across networks that include different types of computer and mobile devices. As well as running on general-purpose computers, software may also have to execute on mobile phones and tablets. You often have to integrate new software with older legacy systems written in different programming languages. The challenge here is to develop techniques for building dependable software that is flexible enough to cope with this heterogeneity.
2. *Business and social change* Businesses and society are changing incredibly quickly as emerging economies develop and new technologies become available. They need to be able to change their existing software and to rapidly develop new software. Many traditional software engineering techniques are time consuming, and delivery of new systems often takes longer than planned. They need to evolve so that the time required for software to deliver value to its customers is reduced.
3. *Security and trust* As software is intertwined with all aspects of our lives, it is essential that we can trust that software. This is especially true for remote software systems accessed through a web page or web service interface. We have to make sure that malicious users cannot successfully attack our software and that information security is maintained.
4. *Scale* Software has to be developed across a very wide range of scales, from very small embedded systems in portable or wearable devices through to Internet-scale, cloud-based systems that serve a global community.

To address these challenges, we will need new tools and techniques as well as innovative ways of combining and using existing software engineering methods.

1.1.2 Software engineering diversity

Software engineering is a systematic approach to the production of software that takes into account practical cost, schedule, and dependability issues, as well as the needs of software customers and producers. The specific methods, tools, and techniques used depend on the organization developing the software, the type of software, and the people involved in the development process. There are no universal software engineering methods that are suitable for all systems and all companies. Rather, a diverse set of software engineering methods and tools has evolved over the past 50 years. However, the SEMAT initiative (Jacobson et al. 2013) proposes that there can be a fundamental meta-process that can be instantiated to create different kinds of process. This is at an early stage of development and may be a basis for improving our current software engineering methods.

Perhaps the most significant factor in determining which software engineering methods and techniques are most important is the type of application being developed. There are many different types of application, including:

1. *Stand-alone applications* These are application systems that run on a personal computer or apps that run on a mobile device. They include all necessary functionality and may not need to be connected to a network. Examples of such applications are office applications on a PC, CAD programs, photo manipulation software, travel apps, productivity apps, and so on.
2. *Interactive transaction-based applications* These are applications that execute on a remote computer and that are accessed by users from their own computers, phones, or tablets. Obviously, these include web applications such as e-commerce applications where you interact with a remote system to buy goods and services. This class of application also includes business systems, where a business provides access to its systems through a web browser or special-purpose client program and cloud-based services, such as mail and photo sharing. Interactive applications often incorporate a large data store that is accessed and updated in each transaction.
3. *Embedded control systems* These are software control systems that control and manage hardware devices. Numerically, there are probably more embedded systems than any other type of system. Examples of embedded systems include the software in a mobile (cell) phone, software that controls antilock braking in a car, and software in a microwave oven to control the cooking process.
4. *Batch processing systems* These are business systems that are designed to process data in large batches. They process large numbers of individual inputs to create corresponding outputs. Examples of batch systems are periodic billing systems, such as phone billing systems, and salary payment systems.
5. *Entertainment systems* These are systems for personal use that are intended to entertain the user. Most of these systems are games of one kind or another, which may run on special-purpose console hardware. The quality of the user interaction offered is the most important distinguishing characteristic of entertainment systems.
6. *Systems for modeling and simulation* These are systems that are developed by scientists and engineers to model physical processes or situations, which include many separate, interacting objects. These are often computationally intensive and require high-performance parallel systems for execution.
7. *Data collection and analysis systems* Data collection systems are systems that collect data from their environment and send that data to other systems for processing. The software may have to interact with sensors and often is installed in a hostile environment such as inside an engine or in a remote location. “Big data” analysis may involve cloud-based systems carrying out statistical analysis and looking for relationships in the collected data.
8. *Systems of systems* These are systems, used in enterprises and other large organizations, that are composed of a number of other software systems. Some of these may be generic software products, such as an ERP system. Other systems in the assembly may be specially written for that environment.

Of course, the boundaries between these system types are blurred. If you develop a game for a phone, you have to take into account the same constraints (power, hardware interaction) as the developers of the phone software. Batch processing systems are often used in conjunction with web-based transaction systems. For example, in a company, travel expense claims may be submitted through a web application but processed in a batch application for monthly payment.

Each type of system requires specialized software engineering techniques because the software has different characteristics. For example, an embedded control system in an automobile is safety-critical and is burned into ROM (read-only memory) when installed in the vehicle. It is therefore very expensive to change. Such a system needs extensive verification and validation so that the chances of having to recall cars after sale to fix software problems are minimized. User interaction is minimal (or perhaps nonexistent), so there is no need to use a development process that relies on user interface prototyping.

For an interactive web-based system or app, iterative development and delivery is the best approach, with the system being composed of reusable components. However, such an approach may be impractical for a system of systems, where detailed specifications of the system interactions have to be specified in advance so that each system can be separately developed.

Nevertheless, there are software engineering fundamentals that apply to all types of software systems:

1. They should be developed using a managed and understood development process. The organization developing the software should plan the development process and have clear ideas of what will be produced and when it will be completed. Of course, the specific process that you should use depends on the type of software that you are developing.
2. Dependability and performance are important for all types of system. Software should behave as expected, without failures, and should be available for use when it is required. It should be safe in its operation and, as far as possible, should be secure against external attack. The system should perform efficiently and should not waste resources.
3. Understanding and managing the software specification and requirements (what the software should do) are important. You have to know what different customers and users of the system expect from it, and you have to manage their expectations so that a useful system can be delivered within budget and to schedule.
4. You should make effective use of existing resources. This means that, where appropriate, you should reuse software that has already been developed rather than write new software.

These fundamental notions of process, dependability, requirements, management, and reuse are important themes of this book. Different methods reflect them in different ways, but they underlie all professional software development.

These fundamentals are independent of the program language used for software development. I don't cover specific programming techniques in this book because these vary dramatically from one type of system to another. For example, a dynamic language, such as Ruby, is the right type of language for interactive system development but is inappropriate for embedded systems engineering.

1.1.3 Internet software engineering

The development of the Internet and the World Wide Web has had a profound effect on all of our lives. Initially, the web was primarily a universally accessible information store, and it had little effect on software systems. These systems ran on local computers and were only accessible from within an organization. Around 2000, the web started to evolve, and more and more functionality was added to browsers. This meant that web-based systems could be developed where, instead of a special-purpose user interface, these systems could be accessed using a web browser. This led to the development of a vast range of new system products that delivered innovative services, accessed over the web. These are often funded by adverts that are displayed on the user's screen and do not involve direct payment from users.

As well as these system products, the development of web browsers that could run small programs and do some local processing led to an evolution in business and organizational software. Instead of writing software and deploying it on users' PCs, the software was deployed on a web server. This made it much cheaper to change and upgrade the software, as there was no need to install the software on every PC. It also reduced costs, as user interface development is particularly expensive. Wherever it has been possible to do so, businesses have moved to web-based interaction with company software systems.

The notion of software as a service (Chapter 17) was proposed early in the 21st century. This has now become the standard approach to the delivery of web-based system products such as Google Apps, Microsoft Office 365, and Adobe Creative Suite. More and more software runs on remote "clouds" instead of local servers and is accessed over the Internet. A computing cloud is a huge number of linked computer systems that is shared by many users. Users do not buy software but pay according to how much the software is used or are given free access in return for watching adverts that are displayed on their screen. If you use services such as web-based mail, storage, or video, you are using a cloud-based system.

The advent of the web has led to a dramatic change in the way that business software is organized. Before the web, business applications were mostly monolithic, single programs running on single computers or computer clusters. Communications were local, within an organization. Now, software is highly distributed, sometimes across the world. Business applications are not programmed from scratch but involve extensive reuse of components and programs.

This change in software organization has had a major effect on software engineering for web-based systems. For example:

1. Software reuse has become the dominant approach for constructing web-based systems. When building these systems, you think about how you can assemble them from preexisting software components and systems, often bundled together in a framework.
2. It is now generally recognized that it is impractical to specify all the requirements for such systems in advance. Web-based systems are always developed and delivered incrementally.
3. Software may be implemented using service-oriented software engineering, where the software components are stand-alone web services. I discuss this approach to software engineering in Chapter 18.
4. Interface development technology such as AJAX (Holdener 2008) and HTML5 (Freeman 2011) have emerged that support the creation of rich interfaces within a web browser.

The fundamental ideas of software engineering, discussed in the previous section, apply to web-based software, as they do to other types of software. Web-based systems are getting larger and larger, so software engineering techniques that deal with scale and complexity are relevant for these systems.

1.2 Software engineering ethics

Like other engineering disciplines, software engineering is carried out within a social and legal framework that limits the freedom of people working in that area. As a software engineer, you must accept that your job involves wider responsibilities than simply the application of technical skills. You must also behave in an ethical and morally responsible way if you are to be respected as a professional engineer.

It goes without saying that you should uphold normal standards of honesty and integrity. You should not use your skills and abilities to behave in a dishonest way or in a way that will bring disrepute to the software engineering profession. However, there are areas where standards of acceptable behavior are not bound by laws but by the more tenuous notion of professional responsibility. Some of these are:

1. *Confidentiality* You should normally respect the confidentiality of your employers or clients regardless of whether or not a formal confidentiality agreement has been signed.
2. *Competence* You should not misrepresent your level of competence. You should not knowingly accept work that is outside your competence.
3. *Intellectual property rights* You should be aware of local laws governing the use of intellectual property such as patents and copyright. You should be careful to ensure that the intellectual property of employers and clients is protected.

Software Engineering Code of Ethics and Professional Practice

ACM/IEEE-CS Joint Task Force on Software Engineering Ethics and Professional Practices

PREAMBLE

The short version of the code summarizes aspirations at a high level of the abstraction; the clauses that are included in the full version give examples and details of how these aspirations change the way we act as software engineering professionals. Without the aspirations, the details can become legalistic and tedious; without the details, the aspirations can become high sounding but empty; together, the aspirations and the details form a cohesive code.

Software engineers shall commit themselves to making the analysis, specification, design, development, testing, and maintenance of software a beneficial and respected profession. In accordance with their commitment to the health, safety, and welfare of the public, software engineers shall adhere to the following Eight Principles:

1. PUBLIC – Software engineers shall act consistently with the public interest.
2. CLIENT AND EMPLOYER – Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.
3. PRODUCT – Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.
4. JUDGMENT – Software engineers shall maintain integrity and independence in their professional judgment.
5. MANAGEMENT – Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.
6. PROFESSION – Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.
7. COLLEAGUES – Software engineers shall be fair to and supportive of their colleagues.
8. SELF – Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

Figure 1.3 The ACM/IEEE Code of Ethics (ACM/IEEE-CS Joint Task Force on Software Engineering Ethics and Professional Practices, short version. <http://www.acm.org/about/se-code>)

(© 1999 by the ACM, Inc. and the IEEE, Inc.)

4. *Computer misuse* You should not use your technical skills to misuse other people's computers. Computer misuse ranges from relatively trivial (game playing on an employer's machine) to extremely serious (dissemination of viruses or other malware).

Professional societies and institutions have an important role to play in setting ethical standards. Organizations such as the ACM, the IEEE (Institute of Electrical and Electronic Engineers), and the British Computer Society publish a code of professional conduct or code of ethics. Members of these organizations undertake to follow that code when they sign up for membership. These codes of conduct are generally concerned with fundamental ethical behavior.

Professional associations, notably the ACM and the IEEE, have cooperated to produce a joint code of ethics and professional practice. This code exists in both a short form, shown in Figure 1.3, and a longer form (Gotterbarn, Miller, and Rogerson 1999) that adds detail and substance to the shorter version. The rationale behind this code is summarized in the first two paragraphs of the longer form:

Computers have a central and growing role in commerce, industry, government, medicine, education, entertainment and society at large. Software engineers are those who contribute by direct participation or by teaching, to the analysis, specification, design, development, certification, maintenance and testing of software systems. Because of their roles in developing software systems, software engineers have significant opportunities to do good or cause harm, to enable others to do good or cause harm, or to influence others to do good or cause harm. To ensure, as much as possible, that their efforts will be used for good, software engineers must commit themselves to making software engineering a beneficial and respected profession. In accordance with that commitment, software engineers shall adhere to the following Code of Ethics and Professional Practice[†].

The Code contains eight Principles related to the behaviour of and decisions made by professional software engineers, including practitioners, educators, managers, supervisors and policy makers, as well as trainees and students of the profession. The Principles identify the ethically responsible relationships in which individuals, groups, and organizations participate and the primary obligations within these relationships. The Clauses of each Principle are illustrations of some of the obligations included in these relationships. These obligations are founded in the software engineer's humanity, in special care owed to people affected by the work of software engineers, and the unique elements of the practice of software engineering. The Code prescribes these as obligations of anyone claiming to be or aspiring to be a software engineer[†].

In any situation where different people have different views and objectives, you are likely to be faced with ethical dilemmas. For example, if you disagree, in principle, with the policies of more senior management in the company, how should you react? Clearly, this depends on the people involved and the nature of the disagreement. Is it best to argue a case for your position from within the organization or to resign in principle? If you feel that there are problems with a software project, when do you reveal these problems to management? If you discuss these while they are just a suspicion, you may be overreacting to a situation; if you leave it too long, it may be impossible to resolve the difficulties.

We all face such ethical dilemmas in our professional lives, and, fortunately, in most cases they are either relatively minor or can be resolved without too much difficulty. Where they cannot be resolved, the engineer is faced with, perhaps, another problem. The principled action may be to resign from their job, but this may well affect others such as their partner or their children.

A difficult situation for professional engineers arises when their employer acts in an unethical way. Say a company is responsible for developing a safety-critical system and, because of time pressure, falsifies the safety validation records. Is the engineer's responsibility to maintain confidentiality or to alert the customer or publicize, in some way, that the delivered system may be unsafe?

[†]ACM/IEEE-CS Joint Task Force on Software Engineering Ethics and Professional Practices, short version Preamble. <http://www.acm.org/about/se-code> Copyright © 1999 by the Association for Computing Machinery, Inc. and the Institute for Electrical and Electronics Engineers, Inc.

The problem here is that there are no absolutes when it comes to safety. Although the system may not have been validated according to predefined criteria, these criteria may be too strict. The system may actually operate safely throughout its lifetime. It is also the case that, even when properly validated, the system may fail and cause an accident. Early disclosure of problems may result in damage to the employer and other employees; failure to disclose problems may result in damage to others.

You must make up your own mind in these matters. The appropriate ethical position here depends on the views of the people involved. The potential for damage, the extent of the damage, and the people affected by the damage should influence the decision. If the situation is very dangerous, it may be justified to publicize it using the national press or social media. However, you should always try to resolve the situation while respecting the rights of your employer.

Another ethical issue is participation in the development of military and nuclear systems. Some people feel strongly about these issues and do not wish to participate in any systems development associated with defense systems. Others will work on military systems but not on weapons systems. Yet others feel that national security is an overriding principle and have no ethical objections to working on weapons systems.

In this situation, it is important that both employers and employees should make their views known to each other in advance. Where an organization is involved in military or nuclear work, it should be able to specify that employees must be willing to accept any work assignment. Equally, if an employee is taken on and makes clear that he or she does not wish to work on such systems, employers should not exert pressure to do so at some later date.

The general area of ethics and professional responsibility is increasingly important as software-intensive systems pervade every aspect of work and everyday life. It can be considered from a philosophical standpoint where the basic principles of ethics are considered and software engineering ethics are discussed with reference to these basic principles. This is the approach taken by Laudon (Laudon 1995) and Johnson (Johnson 2001). More recent texts such as that by Tavani (Tavani 2013) introduce the notion of cyberethics and cover both the philosophical background and practical and legal issues. They include ethical issues for technology users as well as developers.

I find that a philosophical approach is too abstract and difficult to relate to everyday experience so I prefer the more concrete approach embodied in professional codes of conduct (Bott 2005; Duquenoy 2007). I think that ethics are best discussed in a software engineering context and not as a subject in its own right. Therefore, I do not discuss software engineering ethics in an abstract way but include examples in the exercises that can be the starting point for a group discussion.

1.3 Case studies

To illustrate software engineering concepts, I use examples from four different types of system. I have deliberately not used a single case study, as one of the key messages in this book is that software engineering practice depends on the type of systems

being produced. I therefore choose an appropriate example when discussing concepts such as safety and dependability, system modeling, reuse, etc.

The system types that I use as case studies are:

1. *An embedded system* This is a system where the software controls some hardware device and is embedded in that device. Issues in embedded systems typically include physical size, responsiveness, and power management, etc. The example of an embedded system that I use is a software system to control an insulin pump for people who have diabetes.
2. *An information system* The primary purpose of this type of system is to manage and provide access to a database of information. Issues in information systems include security, usability, privacy, and maintaining data integrity. The example of an information system used is a medical records system.
3. *A sensor-based data collection system* This is a system whose primary purposes are to collect data from a set of sensors and to process that data in some way. The key requirements of such systems are reliability, even in hostile environmental conditions, and maintainability. The example of a data collection system that I use is a wilderness weather station.
4. *A support environment.* This is an integrated collection of software tools that are used to support some kind of activity. Programming environments, such as Eclipse (Vogel 2012) will be the most familiar type of environment for readers of this book. I describe an example here of a digital learning environment that is used to support students' learning in schools.

I introduce each of these systems in this chapter; more information about each of them is available on the website (software-engineering-book.com).

1.3.1 An insulin pump control system

An insulin pump is a medical system that simulates the operation of the pancreas (an internal organ). The software controlling this system is an embedded system that collects information from a sensor and controls a pump that delivers a controlled dose of insulin to a user.

People who suffer from diabetes use the system. Diabetes is a relatively common condition in which the human pancreas is unable to produce sufficient quantities of a hormone called insulin. Insulin metabolizes glucose (sugar) in the blood. The conventional treatment of diabetes involves regular injections of genetically engineered insulin. Diabetics measure their blood sugar levels periodically using an external meter and then estimate the dose of insulin they should inject.

The problem is that the level of insulin required does not just depend on the blood glucose level but also on the time of the last insulin injection. Irregular checking can lead to very low levels of blood glucose (if there is too much insulin) or very high levels of blood sugar (if there is too little insulin). Low blood glucose is, in the short term, a more serious condition as it can result in temporary brain malfunctioning and,

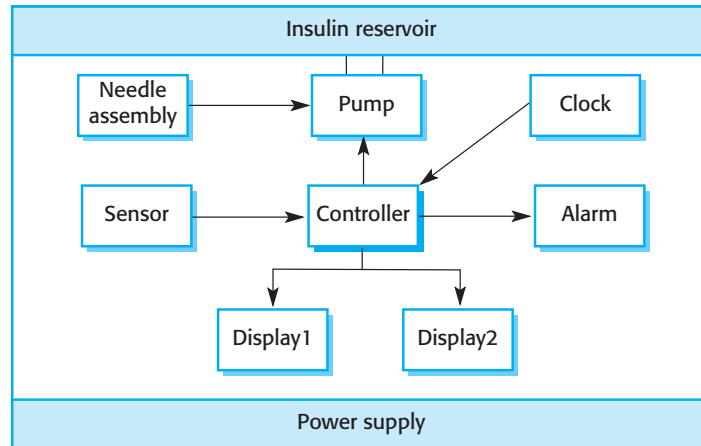


Figure 1.4 Insulin pump hardware architecture

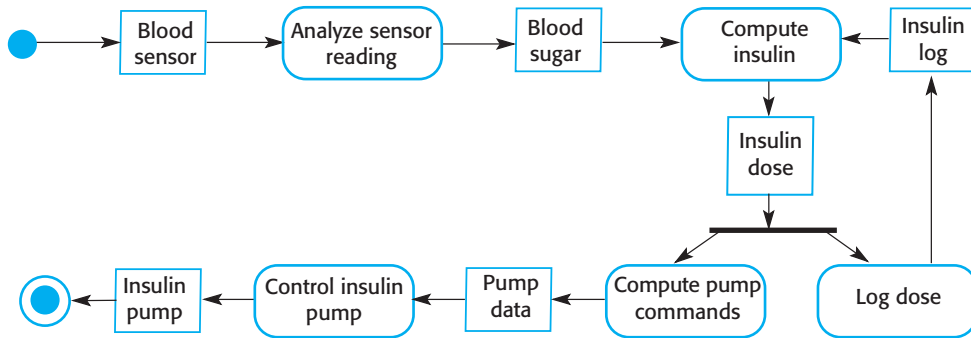


Figure 1.5 Activity model of the insulin pump

ultimately, unconsciousness and death. In the long term, however, continual high levels of blood glucose can lead to eye damage, kidney damage, and heart problems.

Advances in developing miniaturized sensors have meant that it is now possible to develop automated insulin delivery systems. These systems monitor blood sugar levels and deliver an appropriate dose of insulin when required. Insulin delivery systems like this one are now available and are used by patients who find it difficult to control their insulin levels. In future, it may be possible for diabetics to have such systems permanently attached to their bodies.

A software-controlled insulin delivery system uses a microsensor embedded in the patient to measure some blood parameter that is proportional to the sugar level. This is then sent to the pump controller. This controller computes the sugar level and the amount of insulin that is needed. It then sends signals to a miniaturized pump to deliver the insulin via a permanently attached needle.

Figure 1.4 shows the hardware components and organization of the insulin pump. To understand the examples in this book, all you need to know is that the blood sensor measures the electrical conductivity of the blood under different conditions and that these values can be related to the blood sugar level. The insulin pump delivers one unit of insulin in response to a single pulse from a controller. Therefore, to deliver 10 units of insulin, the controller sends 10 pulses to the pump. Figure 1.5 is a Unified Modeling

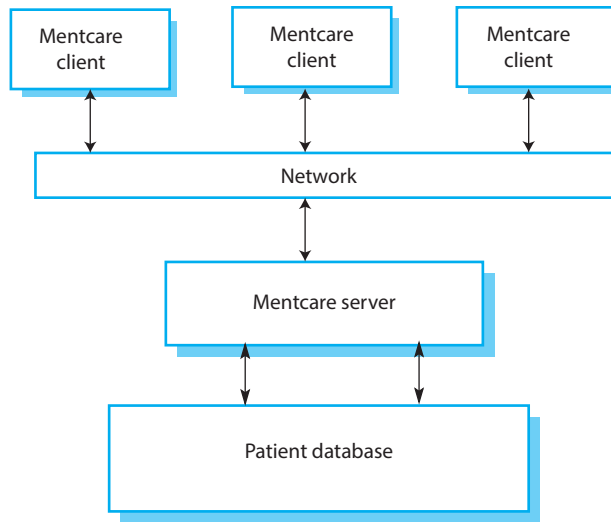


Figure 1.6 The organization of the Mentcare system

Language (UML) activity model that illustrates how the software transforms an input blood sugar level to a sequence of commands that drive the insulin pump.

Clearly, this is a safety-critical system. If the pump fails to operate or does not operate correctly, then the user's health may be damaged or they may fall into a coma because their blood sugar levels are too high or too low. This system must therefore meet two essential high-level requirements:

1. The system shall be available to deliver insulin when required.
2. The system shall perform reliably and deliver the correct amount of insulin to counteract the current level of blood sugar.

The system must therefore be designed and implemented to ensure that it always meets these requirements. More detailed requirements and discussions of how to ensure that the system is safe are discussed in later chapters.

1.3.2 A patient information system for mental health care

A patient information system to support mental health care (the Mentcare system) is a medical information system that maintains information about patients suffering from mental health problems and the treatments that they have received. Most mental health patients do not require dedicated hospital treatment but need to attend specialist clinics regularly where they can meet a doctor who has detailed knowledge of their problems. To make it easier for patients to attend, these clinics are not just run in hospitals. They may also be held in local medical practices or community centers.

The Mentcare system (Figure 1.6) is a patient information system that is intended for use in clinics. It makes use of a centralized database of patient information but

has also been designed to run on a laptop, so that it may be accessed and used from sites that do not have secure network connectivity. When the local systems have secure network access, they use patient information in the database, but they can download and use local copies of patient records when they are disconnected. The system is not a complete medical records system and so does not maintain information about other medical conditions. However, it may interact and exchange data with other clinical information systems.

This system has two purposes:

1. To generate management information that allows health service managers to assess performance against local and government targets.
2. To provide medical staff with timely information to support the treatment of patients.

Patients who suffer from mental health problems are sometimes irrational and disorganized so may miss appointments, deliberately or accidentally lose prescriptions and medication, forget instructions and make unreasonable demands on medical staff. They may drop in on clinics unexpectedly. In a minority of cases, they may be a danger to themselves or to other people. They may regularly change address or may be homeless on a long-term or short-term basis. Where patients are dangerous, they may need to be “sectioned”—that is, confined to a secure hospital for treatment and observation.

Users of the system include clinical staff such as doctors, nurses, and health visitors (nurses who visit people at home to check on their treatment). Nonmedical users include receptionists who make appointments, medical records staff who maintain the records system, and administrative staff who generate reports.

The system is used to record information about patients (name, address, age, next of kin, etc.), consultations (date, doctor seen, subjective impressions of the patient, etc.), conditions, and treatments. Reports are generated at regular intervals for medical staff and health authority managers. Typically, reports for medical staff focus on information about individual patients, whereas management reports are anonymized and are concerned with conditions, costs of treatment, etc.

The key features of the system are:

1. *Individual care management* Clinicians can create records for patients, edit the information in the system, view patient history, and so on. The system supports data summaries so that doctors who have not previously met a patient can quickly learn about the key problems and treatments that have been prescribed.
2. *Patient monitoring* The system regularly monitors the records of patients that are involved in treatment and issues warnings if possible problems are detected. Therefore, if a patient has not seen a doctor for some time, a warning may be issued. One of the most important elements of the monitoring system is to keep track of patients who have been sectioned and to ensure that the legally required checks are carried out at the right time.

3. *Administrative reporting* The system generates monthly management reports showing the number of patients treated at each clinic, the number of patients who have entered and left the care system, the number of patients sectioned, the drugs prescribed and their costs, etc.

Two different laws affect the system: laws on data protection that govern the confidentiality of personal information and mental health laws that govern the compulsory detention of patients deemed to be a danger to themselves or others. Mental health is unique in this respect as it is the only medical speciality that can recommend the detention of patients against their will. This is subject to strict legislative safeguards. One aim of the Mentcare system is to ensure that staff always act in accordance with the law and that their decisions are recorded for judicial review if necessary.

As in all medical systems, privacy is a critical system requirement. It is essential that patient information is confidential and is never disclosed to anyone apart from authorized medical staff and the patient themselves. The Mentcare system is also a safety-critical system. Some mental illnesses cause patients to become suicidal or a danger to other people. Wherever possible, the system should warn medical staff about potentially suicidal or dangerous patients.

The overall design of the system has to take into account privacy and safety requirements. The system must be available when needed; otherwise safety may be compromised, and it may be impossible to prescribe the correct medication to patients. There is a potential conflict here. Privacy is easiest to maintain when there is only a single copy of the system data. However, to ensure availability in the event of server failure or when disconnected from a network, multiple copies of the data should be maintained. I discuss the trade-offs between these requirements in later chapters.

1.3.3 A wilderness weather station

To help monitor climate change and to improve the accuracy of weather forecasts in remote areas, the government of a country with large areas of wilderness decides to deploy several hundred weather stations in remote areas. These weather stations collect data from a set of instruments that measure temperature and pressure, sunshine, rainfall, wind speed and wind direction.

Wilderness weather stations are part of a larger system (Figure 1.7), which is a weather information system that collects data from weather stations and makes it available to other systems for processing. The systems in Figure 1.7 are:

1. *The weather station system* This system is responsible for collecting weather data, carrying out some initial data processing, and transmitting it to the data management system.
2. *The data management and archiving system* This system collects the data from all of the wilderness weather stations, carries out data processing and analysis, and archives the data in a form that can be retrieved by other systems, such as weather forecasting systems.

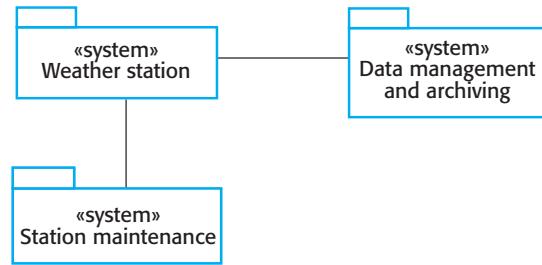


Figure 1.7 The weather station's environment

3. *The station maintenance system* This system can communicate by satellite with all wilderness weather stations to monitor the health of these systems and provide reports of problems. It can update the embedded software in these systems. In the event of system problems, this system can also be used to remotely control the weather station.

In Figure 1.7, I have used the UML package symbol to indicate that each system is a collection of components and the separate systems are identified using the UML stereotype «system». The associations between the packages indicate there is an exchange of information but, at this stage, there is no need to define them in any more detail.

The weather stations include instruments that measure weather parameters such as wind speed and direction, ground and air temperatures, barometric pressure, and rainfall over a 24-hour period. Each of these instruments is controlled by a software system that takes parameter readings periodically and manages the data collected from the instruments.

The weather station system operates by collecting weather observations at frequent intervals; for example, temperatures are measured every minute. However, because the bandwidth to the satellite is relatively narrow, the weather station carries out some local processing and aggregation of the data. It then transmits this aggregated data when requested by the data collection system. If it is impossible to make a connection, then the weather station maintains the data locally until communication can be resumed.

Each weather station is battery-powered and must be entirely self-contained; there are no external power or network cables. All communications are through a relatively slow satellite link, and the weather station must include some mechanism (solar or wind power) to charge its batteries. As they are deployed in wilderness areas, they are exposed to severe environmental conditions and may be damaged by animals. The station software is therefore not just concerned with data collection. It must also:

1. Monitor the instruments, power, and communication hardware and report faults to the management system.
2. Manage the system power, ensuring that batteries are charged whenever the environmental conditions permit but also that generators are shut down in potentially damaging weather conditions, such as high wind.

3. Allow for dynamic reconfiguration where parts of the software are replaced with new versions and where backup instruments are switched into the system in the event of system failure.

Because weather stations have to be self-contained and unattended, this means that the software installed is complex, even though the data collection functionality is fairly simple.

1.3.4 A digital learning environment for schools

Many teachers argue that using interactive software systems to support education can lead to both improved learner motivation and a deeper level of knowledge and understanding in students. However, there is no general agreement on the ‘best’ strategy for computer-supported learning, and teachers in practice use a range of different interactive, web-based tools to support learning. The tools used depend on the ages of the learners, their cultural background, their experience with computers, equipment available, and the preferences of the teachers involved.

A digital learning environment is a framework in which a set of general-purpose and specially designed tools for learning may be embedded, plus a set of applications that are geared to the needs of the learners using the system. The framework provides general services such as an authentication service, synchronous and asynchronous communication services, and a storage service.

The tools included in each version of the environment are chosen by teachers and learners to suit their specific needs. These can be general applications such as spreadsheets, learning management applications such as a Virtual Learning Environment (VLE) to manage homework submission and assessment, games, and simulations. They may also include specific content, such as content about the American Civil War and applications to view and annotate that content.

Figure 1.8 is a high-level architectural model of a digital learning environment (iLearn) that was designed for use in schools for students from 3 to 18 years of age. The approach adopted is that this is a distributed system in which all components of the environment are services that can be accessed from anywhere on the Internet. There is no requirement that all of the learning tools are gathered together in one place.

The system is a service-oriented system with all system components considered to be a replaceable service. There are three types of service in the system:

1. *Utility services* that provide basic application-independent functionality and that may be used by other services in the system. Utility services are usually developed or adapted specifically for this system.
2. *Application services* that provide specific applications such as email, conferencing, photo sharing, etc., and access to specific educational content such as scientific films or historical resources. Application services are external services that are either specifically purchased for the system or are available freely over the Internet.

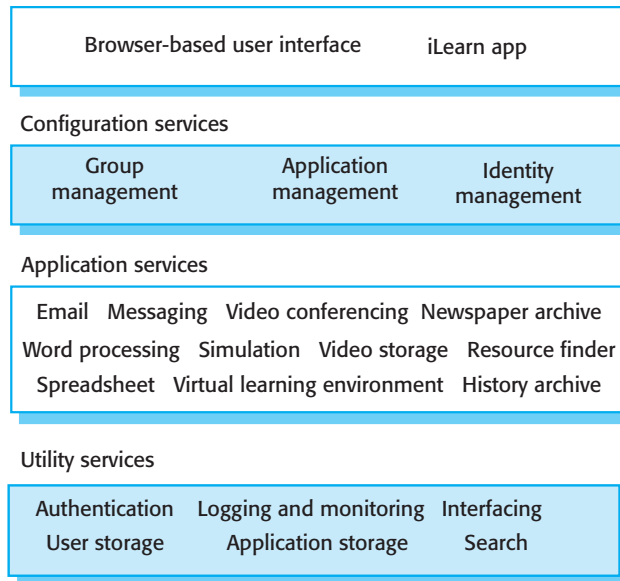


Figure 1.8 The architecture of a digital learning environment (iLearn)

3. *Configuration services* that are used to adapt the environment with a specific set of application services and to define how services are shared between students, teachers, and their parents.

The environment has been designed so that services can be replaced as new services become available and to provide different versions of the system that are suited for the age of the users. This means that the system has to support two levels of service integration:

1. *Integrated services* are services that offer an API (application programming interface) and that can be accessed by other services through that API. Direct service-to-service communication is therefore possible. An authentication service is an example of an integrated service. Rather than use their own authentication mechanisms, an authentication service may be called on by other services to authenticate users. If users are already authenticated, then the authentication service may pass authentication information directly to another service, via an API, with no need for users to reauthenticate themselves.
2. *Independent services* are services that are simply accessed through a browser interface and that operate independently of other services. Information can only be shared with other services through explicit user actions such as copy and paste; reauthentication may be required for each independent service.

If an independent service becomes widely used, the development team may then integrate that service so that it becomes an integrated and supported service.

KEY POINTS

- Software engineering is an engineering discipline that is concerned with all aspects of software production.
- Software is not just a program or programs but also includes all electronic documentation that is needed by system users, quality assurance staff, and developers. Essential software product attributes are maintainability, dependability and security, efficiency, and acceptability.
- The software process includes all of the activities involved in software development. The high-level activities of specification, development, validation, and evolution are part of all software processes.
- There are many different types of system, and each requires appropriate software engineering tools and techniques for their development. Few, if any, specific design and implementation techniques are applicable to all kinds of system.
- The fundamental ideas of software engineering are applicable to all types of software system. These fundamentals include managed software processes, software dependability and security, requirements engineering, and software reuse.
- Software engineers have responsibilities to the engineering profession and society. They should not simply be concerned with technical issues but should be aware of the ethical issues that affect their work.
- Professional societies publish codes of conduct that embed ethical and professional standards. These set out the standards of behavior expected of their members.

FURTHER READING

“Software Engineering Code of Ethics Is Approved.” An article that discusses the background to the development of the ACM/IEEE Code of Ethics and that includes both the short and long form of the code. (*Comm. ACM*, D. Gotterbarn, K. Miller, and S. Rogerson, October 1999). <http://dx.doi.org/10.1109/MC.1999.796142>

“A View of 20th and 21st Century Software Engineering.” A backward and forward look at software engineering from one of the first and most distinguished software engineers. Barry Boehm identifies timeless software engineering principles but also suggests that some commonly used practices are obsolete. (B. Boehm, *Proc. 28th Software Engineering Conf.*, Shanghai, 2006). <http://dx.doi.org/10.1145/1134285.1134288>

“Software Engineering Ethics.” Special issue of *IEEE Computer*, with several papers on the topic (*IEEE Computer*, 42 (6), June 2009).

Ethics for the Information Age. This is a wide-ranging book that covers all aspects of information technology (IT) ethics, not simply ethics for software engineers. I think this is the right approach as you really need to understand software engineering ethics within a wider ethical framework (M. J. Quinn, 2013, Addison-Wesley).

The Essence of Software Engineering: Applying the SEMAT kernel. This book discusses the idea of a universal framework that can underlie all software engineering methods. It can be adapted and used for all types of systems and organizations. I am personally skeptical about whether or not a universal approach is realistic in practice, but the book has some interesting ideas that are worth exploring. (I. Jacobsen, P-W Ng, P. E. McMahon, I. Spence, and S. Lidman, 2013, Addison-Wesley)

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

<http://software-engineering-book.com/videos/software-engineering/>

Links to case study descriptions:

<http://software-engineering-book.com/case-studies/>

EXERCISES

- 1.1. Explain why professional software that is developed for a customer is not simply the programs that have been developed and delivered.
- 1.2. What is the most important difference between generic software product development and custom software development? What might this mean in practice for users of generic software products?
- 1.3. Briefly discuss why it is usually cheaper in the long run to use software engineering methods and techniques for software systems.
- 1.4. Software engineering is not only concerned with issues like system heterogeneity, business and social change, trust, and security, but also with ethical issues affecting the domain. Give some examples of ethical issues that have an impact on the software engineering domain.
- 1.5. Based on your own knowledge of some of the application types discussed in Section 1.1.2, explain, with examples, why different application types require specialized software engineering techniques to support their design and development.
- 1.6. Explain why the fundamental software engineering principles of process, dependability, requirements management, and reuse are relevant to all types of software system.
- 1.7. Explain how electronic connectivity between various development teams can support software engineering activities.
- 1.8. Noncertified individuals are still allowed to practice software engineering. Discuss some of the possible drawbacks of this.

- 1.9. For each of the clauses in the ACM/IEEE Code of Ethics shown in Figure 1.4, propose an appropriate example that illustrates that clause.
- 1.10. The “Drone Revolution” is currently being debated and discussed all over the world. Drones are unmanned flying machines that are built and equipped with various kinds of software systems that allow them to see, hear, and act. Discuss some of the societal challenges of building such kinds of systems.

REFERENCES

- Bott, F. 2005. *Professional Issues in Information Technology*. Swindon, UK: British Computer Society.
- Duquenoy, P. 2007. *Ethical, Legal and Professional Issues in Computing*. London: Thomson Learning.
- Freeman, A. 2011. *The Definitive Guide to HTML5*. New York: Apress.
- Gotterbarn, D., K. Miller, and S. Rogerson. 1999. “Software Engineering Code of Ethics Is Approved.” *Comm. ACM* 42 (10): 102–107. doi:10.1109/MC.1999.796142.
- Holdener, A. T. 2008. *Ajax: The Definitive Guide*. Sebastopol, CA: O’Reilly and Associates.
- Jacobson, I., P-W. Ng, P. E. McMahon, I. Spence, and S. Lidman. 2013. *The Essence of Software Engineering*. Boston: Addison-Wesley.
- Johnson, D. G. 2001. *Computer Ethics*. Englewood Cliffs, NJ: Prentice-Hall.
- Laudon, K. 1995. “Ethical Concepts and Information Technology.” *Comm. ACM* 38 (12): 33–39. doi:10.1145/219663.219677.
- Naur, P., and Randell, B. 1969. Software Engineering: Report on a conference sponsored by the NATO Science Committee. Brussels. <http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.pdf>
- Tavani, H. T. 2013. *Ethics and Technology: Controversies, Questions, and Strategies for Ethical Computing, 4th ed.* New York: John Wiley & Sons.
- Vogel, L. 2012. *Eclipse 4 Application Development: The Complete Guide to Eclipse 4 RCP Development*. Sebastopol, CA: O’Reilly & Associates.



2

Software processes

Objectives

The objective of this chapter is to introduce you to the idea of a software process—a coherent set of activities for software production. When you have read this chapter, you will:

- understand the concepts of software processes and software process models;
- have been introduced to three general software process models and when they might be used;
- know about the fundamental process activities of software requirements engineering, software development, testing, and evolution;
- understand why processes should be organized to cope with changes in the software requirements and design;
- understand the notion of software process improvement and the factors that affect software process quality.

Contents

- 2.1** Software process models
- 2.2** Process activities
- 2.3** Coping with change
- 2.4** Process improvement

A software process is a set of related activities that leads to the production of a software system. As I discussed in Chapter 1, there are many different types of software systems, and there is no universal software engineering method that is applicable to all of them. Consequently, there is no universally applicable software process. The process used in different companies depends on the type of software being developed, the requirements of the software customer, and the skills of the people writing the software.

However, although there are many different software processes, they all must include, in some form, the four fundamental software engineering activities that I introduced in Chapter 1:

1. *Software specification* The functionality of the software and constraints on its operation must be defined.
2. *Software development* The software to meet the specification must be produced.
3. *Software validation* The software must be validated to ensure that it does what the customer wants.
4. *Software evolution* The software must evolve to meet changing customer needs.

These activities are complex activities in themselves, and they include subactivities such as requirements validation, architectural design, and unit testing. Processes also include other activities, such as software configuration management and project planning that support production activities.

When we describe and discuss processes, we usually talk about the activities in these processes, such as specifying a data model and designing a user interface, and the ordering of these activities. We can all relate to what people do to develop software. However, when describing processes, it is also important to describe who is involved, what is produced, and conditions that influence the sequence of activities:

1. Products or deliverables are the outcomes of a process activity. For example, the outcome of the activity of architectural design may be a model of the software architecture.
2. Roles reflect the responsibilities of the people involved in the process. Examples of roles are project manager, configuration manager, and programmer.
3. Pre- and postconditions are conditions that must hold before and after a process activity has been enacted or a product produced. For example, before architectural design begins, a precondition may be that the consumer has approved all requirements; after this activity is finished, a postcondition might be that the UML models describing the architecture have been reviewed.

Software processes are complex and, like all intellectual and creative processes, rely on people making decisions and judgments. As there is no universal process that is right for all kinds of software, most software companies have developed their own

development processes. Processes have evolved to take advantage of the capabilities of the software developers in an organization and the characteristics of the systems that are being developed. For safety-critical systems, a very structured development process is required where detailed records are maintained. For business systems, with rapidly changing requirements, a more flexible, agile process is likely to be better.

As I discussed in Chapter 1, professional software development is a managed activity, so planning is an inherent part of all processes. Plan-driven processes are processes where all of the process activities are planned in advance and progress is measured against this plan. In agile processes, which I discuss in Chapter 3, planning is incremental and continual as the software is developed. It is therefore easier to change the process to reflect changing customer or product requirements. As Boehm and Turner (Boehm and Turner 2004) explain, each approach is suitable for different types of software. Generally, for large systems, you need to find a balance between plan-driven and agile processes.

Although there is no universal software process, there is scope for process improvement in many organizations. Processes may include outdated techniques or may not take advantage of the best practice in industrial software engineering. Indeed, many organizations still do not take advantage of software engineering methods in their software development. They can improve their process by introducing techniques such as UML modeling and test-driven development. I discuss software process improvement briefly later in this chapter text and in more detail in web Chapter 26.

2.1 Software process models

As I explained in Chapter 1, a software process model (sometimes called a Software Development Life Cycle or SDLC model) is a simplified representation of a software process. Each process model represents a process from a particular perspective and thus only provides partial information about that process. For example, a process activity model shows the activities and their sequence but may not show the roles of the people involved in these activities. In this section, I introduce a number of very general process models (sometimes called *process paradigms*) and present these from an architectural perspective. That is, we see the framework of the process but not the details of process activities.

These generic models are high-level, abstract descriptions of software processes that can be used to explain different approaches to software development. You can think of them as process frameworks that may be extended and adapted to create more specific software engineering processes.

The general process models that I cover here are:

1. *The waterfall model* This takes the fundamental process activities of specification, development, validation, and evolution and represents them as separate process phases such as requirements specification, software design, implementation, and testing.



The Rational Unified Process

The Rational Unified Process (RUP) brings together elements of all of the general process models discussed here and supports prototyping and incremental delivery of software (Krutchen 2003). The RUP is normally described from three perspectives: a dynamic perspective that shows the phases of the model in time, a static perspective that shows process activities, and a practice perspective that suggests good practices to be used in the process. Phases of the RUP are inception, where a business case for the system is established; elaboration, where requirements and architecture are developed; construction where the software is implemented; and transition, where the system is deployed.

<http://software-engineering-book.com/web/rup/>

2. *Incremental development* This approach interleaves the activities of specification, development, and validation. The system is developed as a series of versions (increments), with each version adding functionality to the previous version.
3. *Integration and configuration* This approach relies on the availability of reusable components or systems. The system development process focuses on configuring these components for use in a new setting and integrating them into a system.

As I have said, there is no universal process model that is right for all kinds of software development. The right process depends on the customer and regulatory requirements, the environment where the software will be used, and the type of software being developed. For example, safety-critical software is usually developed using a waterfall process as lots of analysis and documentation is required before implementation begins. Software products are now always developed using an incremental process model. Business systems are increasingly being developed by configuring existing systems and integrating these to create a new system with the functionality that is required.

The majority of practical software processes are based on a general model but often incorporate features of other models. This is particularly true for large systems engineering. For large systems, it makes sense to combine some of the best features of all of the general processes. You need to have information about the essential system requirements to design a software architecture to support these requirements. You cannot develop this incrementally. Subsystems within a larger system may be developed using different approaches. Parts of the system that are well understood can be specified and developed using a waterfall-based process or may be bought in as off-the-shelf systems for configuration. Other parts of the system, which are difficult to specify in advance, should always be developed using an incremental approach. In both cases, software components are likely to be reused.

Various attempts have been made to develop “universal” process models that draw on all of these general models. One of the best known of these universal models is the Rational Unified Process (RUP) (Krutchen 2003), which was developed by Rational, a U.S. software engineering company. The RUP is a flexible model that

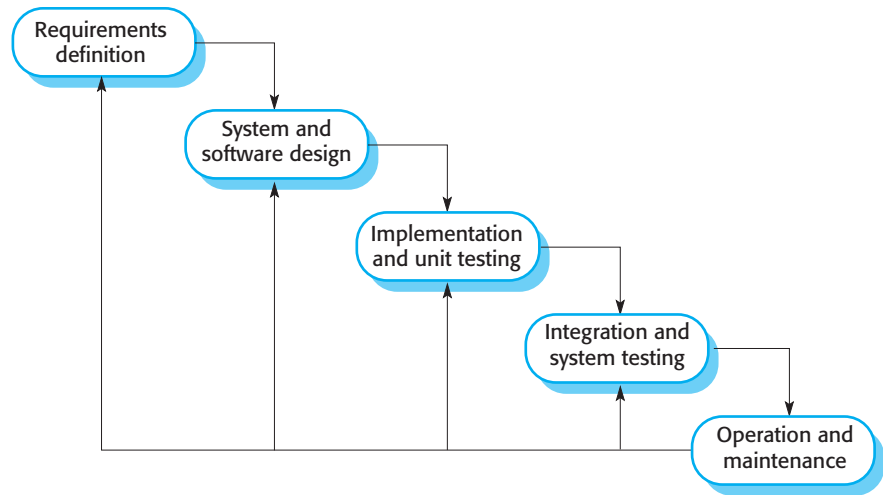


Figure 2.1 The waterfall model

can be instantiated in different ways to create processes that resemble any of the general process models discussed here. The RUP has been adopted by some large software companies (notably IBM), but it has not gained widespread acceptance.

2.1.1 The waterfall model

The first published model of the software development process was derived from engineering process models used in large military systems engineering (Royce 1970). It presents the software development process as a number of stages, as shown in Figure 2.1. Because of the cascade from one phase to another, this model is known as the waterfall model or software life cycle. The waterfall model is an example of a plan-driven process. In principle at least, you plan and schedule all of the process activities before starting software development.

The stages of the waterfall model directly reflect the fundamental software development activities:

1. *Requirements analysis and definition* The system's services, constraints, and goals are established by consultation with system users. They are then defined in detail and serve as a system specification.
2. *System and software design* The systems design process allocates the requirements to either hardware or software systems. It establishes an overall system architecture. Software design involves identifying and describing the fundamental software system abstractions and their relationships.
3. *Implementation and unit testing* During this stage, the software design is realized as a set of programs or program units. Unit testing involves verifying that each unit meets its specification.



Boehm's spiral process model

Barry Boehm, one of the pioneers in software engineering, proposed an incremental process model that was risk-driven. The process is represented as a spiral rather than a sequence of activities (Boehm 1988).

Each loop in the spiral represents a phase of the software process. Thus, the innermost loop might be concerned with system feasibility, the next loop with requirements definition, the next loop with system design, and so on. The spiral model combines change avoidance with change tolerance. It assumes that changes are a result of project risks and includes explicit risk management activities to reduce these risks.

<http://software-engineering-book.com/web/spiral-model/>

4. *Integration and system testing* The individual program units or programs are integrated and tested as a complete system to ensure that the software requirements have been met. After testing, the software system is delivered to the customer.
5. *Operation and maintenance* Normally, this is the longest life-cycle phase. The system is installed and put into practical use. Maintenance involves correcting errors that were not discovered in earlier stages of the life cycle, improving the implementation of system units, and enhancing the system's services as new requirements are discovered.

In principle, the result of each phase in the waterfall model is one or more documents that are approved ("signed off"). The following phase should not start until the previous phase has finished. For hardware development, where high manufacturing costs are involved, this makes sense. However, for software development, these stages overlap and feed information to each other. During design, problems with requirements are identified; during coding design problems are found, and so on. The software process, in practice, is never a simple linear model but involves feedback from one phase to another.

As new information emerges in a process stage, the documents produced at previous stages should be modified to reflect the required system changes. For example, if it is discovered that a requirement is too expensive to implement, the requirements document should be changed to remove that requirement. However, this requires customer approval and delays the overall development process.

As a result, both customers and developers may prematurely freeze the software specification so that no further changes are made to it. Unfortunately, this means that problems are left for later resolution, ignored, or programmed around. Premature freezing of requirements may mean that the system won't do what the user wants. It may also lead to badly structured systems as design problems are circumvented by implementation tricks.

During the final life-cycle phase (operation and maintenance) the software is put into use. Errors and omissions in the original software requirements are discovered.

Program and design errors emerge, and the need for new functionality is identified. The system must therefore evolve to remain useful. Making these changes (software maintenance) may involve repeating previous process stages.

In reality, software has to be flexible and accommodate change as it is being developed. The need for early commitment and system rework when changes are made means that the waterfall model is only appropriate for some types of system:

1. Embedded systems where the software has to interface with hardware systems. Because of the inflexibility of hardware, it is not usually possible to delay decisions on the software's functionality until it is being implemented.
2. Critical systems where there is a need for extensive safety and security analysis of the software specification and design. In these systems, the specification and design documents must be complete so that this analysis is possible. Safety-related problems in the specification and design are usually very expensive to correct at the implementation stage.
3. Large software systems that are part of broader engineering systems developed by several partner companies. The hardware in the systems may be developed using a similar model, and companies find it easier to use a common model for hardware and software. Furthermore, where several companies are involved, complete specifications may be needed to allow for the independent development of different subsystems.

The waterfall model is not the right process model in situations where informal team communication is possible and software requirements change quickly. Iterative development and agile methods are better for these systems.

An important variant of the waterfall model is formal system development, where a mathematical model of a system specification is created. This model is then refined, using mathematical transformations that preserve its consistency, into executable code. Formal development processes, such as that based on the B method (Abrial 2005, 2010), are mostly used in the development of software systems that have stringent safety, reliability, or security requirements. The formal approach simplifies the production of a safety or security case. This demonstrates to customers or regulators that the system actually meets its safety or security requirements. However, because of the high costs of developing a formal specification, this development model is rarely used except for critical systems engineering.

2.1.2 Incremental development

Incremental development is based on the idea of developing an initial implementation, getting feedback from users and others, and evolving the software through several versions until the required system has been developed (Figure 2.2). Specification, development, and validation activities are interleaved rather than separate, with rapid feedback across activities.

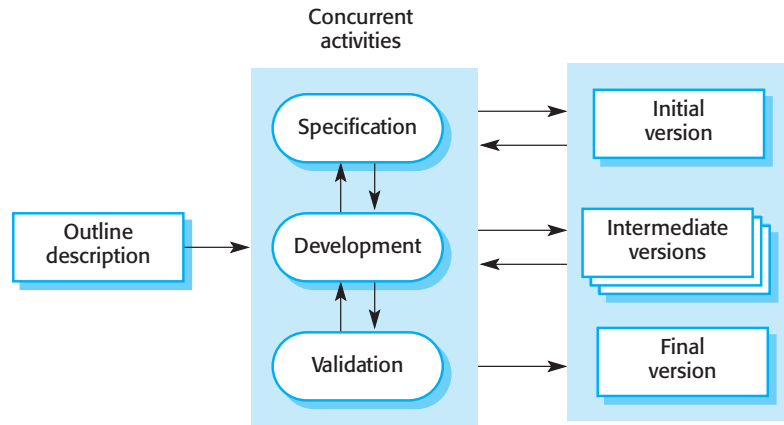


Figure 2.2 Incremental development

Incremental development in some form is now the most common approach for the development of application systems and software products. This approach can be either plan-driven, agile or, more usually, a mixture of these approaches. In a plan-driven approach, the system increments are identified in advance; if an agile approach is adopted, the early increments are identified, but the development of later increments depends on progress and customer priorities.

Incremental software development, which is a fundamental part of agile development methods, is better than a waterfall approach for systems whose requirements are likely to change during the development process. This is the case for most business systems and software products. Incremental development reflects the way that we solve problems. We rarely work out a complete problem solution in advance but move toward a solution in a series of steps, backtracking when we realize that we have made a mistake. By developing the software incrementally, it is cheaper and easier to make changes in the software as it is being developed.

Each increment or version of the system incorporates some of the functionality that is needed by the customer. Generally, the early increments of the system include the most important or most urgently required functionality. This means that the customer or user can evaluate the system at a relatively early stage in the development to see if it delivers what is required. If not, then only the current increment has to be changed and, possibly, new functionality defined for later increments.

Incremental development has three major advantages over the waterfall model:

1. The cost of implementing requirements changes is reduced. The amount of analysis and documentation that has to be redone is significantly less than is required with the waterfall model.
2. It is easier to get customer feedback on the development work that has been done. Customers can comment on demonstrations of the software and see how



Problems with incremental development

Although incremental development has many advantages, it is not problem free. The primary cause of the difficulty is the fact that large organizations have bureaucratic procedures that have evolved over time and there may be a mismatch between these procedures and a more informal iterative or agile process.

Sometimes these procedures are there for good reasons. For example, there may be procedures to ensure that the software meets properly implements external regulations (e.g., in the United States, the Sarbanes Oxley accounting regulations). Changing these procedures may not be possible, so process conflicts may be unavoidable.

<http://software-engineering-book.com/web/incremental-development/>

much has been implemented. Customers find it difficult to judge progress from software design documents.

3. Early delivery and deployment of useful software to the customer is possible, even if all of the functionality has not been included. Customers are able to use and gain value from the software earlier than is possible with a waterfall process.

From a management perspective, the incremental approach has two problems:

1. The process is not visible. Managers need regular deliverables to measure progress. If systems are developed quickly, it is not cost effective to produce documents that reflect every version of the system.
2. System structure tends to degrade as new increments are added. Regular change leads to messy code as new functionality is added in whatever way is possible. It becomes increasingly difficult and costly to add new features to a system. To reduce structural degradation and general code messiness, agile methods suggest that you should regularly refactor (improve and restructure) the software.

The problems of incremental development become particularly acute for large, complex, long-lifetime systems, where different teams develop different parts of the system. Large systems need a stable framework or architecture, and the responsibilities of the different teams working on parts of the system need to be clearly defined with respect to that architecture. This has to be planned in advance rather than developed incrementally.

Incremental development does not mean that you have to deliver each increment to the system customer. You can develop a system incrementally and expose it to customers and other stakeholders for comment, without necessarily delivering it and deploying it in the customer's environment. Incremental delivery (covered in Section 2.3.2) means that the software is used in real, operational processes, so user feedback is likely to be realistic. However, providing feedback is not always possible as experimenting with new software can disrupt normal business processes.

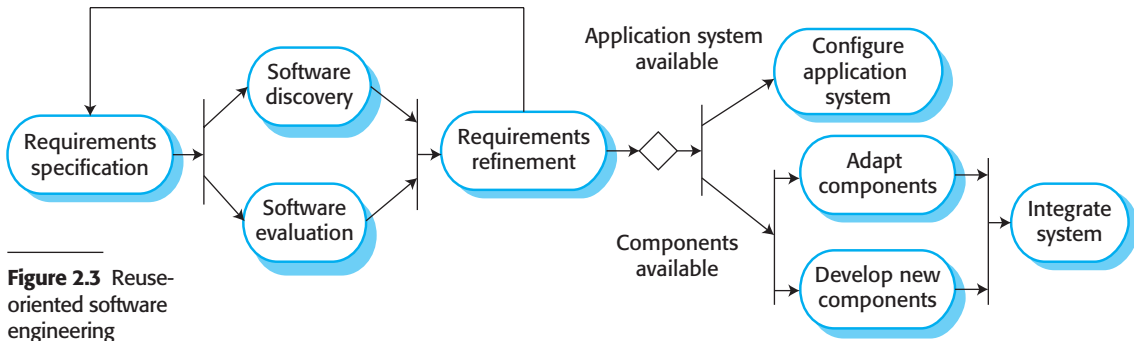


Figure 2.3 Reuse-oriented software engineering

2.1.3 Integration and configuration

In the majority of software projects, there is some software reuse. This often happens informally when people working on the project know of or search for code that is similar to what is required. They look for these, modify them as needed, and integrate them with the new code that they have developed.

This informal reuse takes place regardless of the development process that is used. However, since 2000, software development processes that focus on the reuse of existing software have become widely used. Reuse-oriented approaches rely on a base of reusable software components and an integrating framework for the composition of these components.

Three types of software components are frequently reused:

1. Stand-alone application systems that are configured for use in a particular environment. These systems are general-purpose systems that have many features, but they have to be adapted for use in a specific application.
2. Collections of objects that are developed as a component or as a package to be integrated with a component framework such as the Java Spring framework (Wheeler and White 2013).
3. Web services that are developed according to service standards and that are available for remote invocation over the Internet.

Figure 2.3 shows a general process model for reuse-based development, based on integration and configuration. The stages in this process are:

1. *Requirements specification* The initial requirements for the system are proposed. These do not have to be elaborated in detail but should include brief descriptions of essential requirements and desirable system features.
2. *Software discovery and evaluation* Given an outline of the software requirements, a search is made for components and systems that provide the functionality required. Candidate components and systems are evaluated to see if



Software development tools

Software development tools are programs that are used to support software engineering process activities. These tools include requirements management tools, design editors, refactoring support tools, compilers, debuggers, bug trackers, and system building tools.

Software tools provide process support by automating some process activities and by providing information about the software that is being developed. For example:

- The development of graphical system models as part of the requirements specification or the software design
- The generation of code from these graphical models
- The generation of user interfaces from a graphical interface description that is created interactively by the user
- Program debugging through the provision of information about an executing program
- The automated translation of programs written using an old version of a programming language to a more recent version

Tools may be combined within a framework called an Interactive Development Environment or IDE. This provides a common set of facilities that tools can use so that it is easier for tools to communicate and operate in an integrated way.

<http://software-engineering-book.com/web/software-tools/>

they meet the essential requirements and if they are generally suitable for use in the system.

3. *Requirements refinement* During this stage, the requirements are refined using information about the reusable components and applications that have been discovered. The requirements are modified to reflect the available components, and the system specification is re-defined. Where modifications are impossible, the component analysis activity may be reentered to search for alternative solutions.
4. *Application system configuration* If an off-the-shelf application system that meets the requirements is available, it may then be configured for use to create the new system.
5. *Component adaptation and integration* If there is no off-the-shelf system, individual reusable components may be modified and new components developed. These are then integrated to create the system.

Reuse-oriented software engineering, based around configuration and integration, has the obvious advantage of reducing the amount of software to be developed and so reducing cost and risks. It usually also leads to faster delivery of the software. However, requirements compromises are inevitable, and this may lead to a system

that does not meet the real needs of users. Furthermore, some control over the system evolution is lost as new versions of the reusable components are not under the control of the organization using them.

Software reuse is very important, and so several chapters in the third I have dedicated several chapters in the 3rd part of the book to this topic. General issues of software reuse are covered in Chapter 15, component-based software engineering in Chapters 16 and 17, and service-oriented systems in Chapter 18.

2.2 Process activities

Real software processes are interleaved sequences of technical, collaborative, and managerial activities with the overall goal of specifying, designing, implementing, and testing a software system. Generally, processes are now tool-supported. This means that software developers may use a range of software tools to help them, such as requirements management systems, design model editors, program editors, automated testing tools, and debuggers.

The four basic process activities of specification, development, validation, and evolution are organized differently in different development processes. In the waterfall model, they are organized in sequence, whereas in incremental development they are interleaved. How these activities are carried out depends on the type of software being developed, the experience and competence of the developers, and the type of organization developing the software.

2.2.1 Software specification

Software specification or requirements engineering is the process of understanding and defining what services are required from the system and identifying the constraints on the system's operation and development. Requirements engineering is a particularly critical stage of the software process, as mistakes made at this stage inevitably lead to later problems in the system design and implementation.

Before the requirements engineering process starts, a company may carry out a feasibility or marketing study to assess whether or not there is a need or a market for the software and whether or not it is technically and financially realistic to develop the software required. Feasibility studies are short-term, relatively cheap studies that inform the decision of whether or not to go ahead with a more detailed analysis.

The requirements engineering process (Figure 2.4) aims to produce an agreed requirements document that specifies a system satisfying stakeholder requirements. Requirements are usually presented at two levels of detail. End-users and customers need a high-level statement of the requirements; system developers need a more detailed system specification.

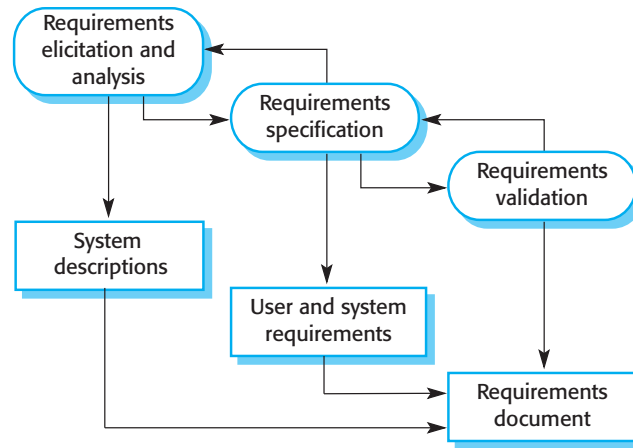


Figure 2.4 The requirements engineering process

There are three main activities in the requirements engineering process:

1. *Requirements elicitation and analysis* This is the process of deriving the system requirements through observation of existing systems, discussions with potential users and procurers, task analysis, and so on. This may involve the development of one or more system models and prototypes. These help you understand the system to be specified.
2. *Requirements specification* Requirements specification is the activity of translating the information gathered during requirements analysis into a document that defines a set of requirements. Two types of requirements may be included in this document. User requirements are abstract statements of the system requirements for the customer and end-user of the system; system requirements are a more detailed description of the functionality to be provided.
3. *Requirements validation* This activity checks the requirements for realism, consistency, and completeness. During this process, errors in the requirements document are inevitably discovered. It must then be modified to correct these problems.

Requirements analysis continues during definition and specification, and new requirements come to light throughout the process. Therefore, the activities of analysis, definition, and specification are interleaved.

In agile methods, requirements specification is not a separate activity but is seen as part of system development. Requirements are informally specified for each increment of the system just before that increment is developed. Requirements are specified according to user priorities. The elicitation of requirements comes from users who are part of or work closely with the development team.

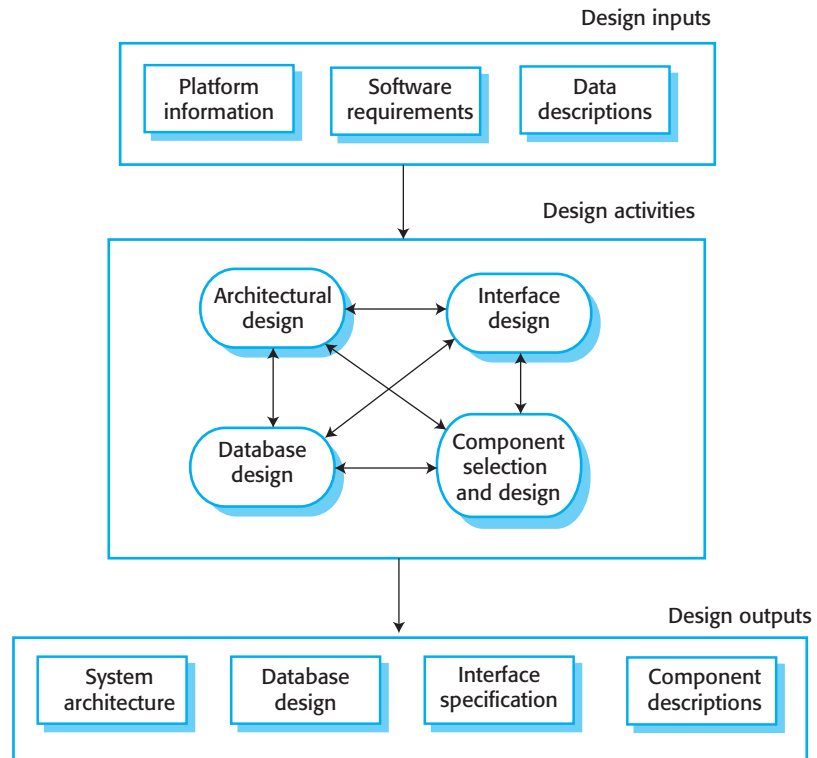


Figure 2.5 A general model of the design process

2.2.2 Software design and implementation

The implementation stage of software development is the process of developing an executable system for delivery to the customer. Sometimes this involves separate activities of software design and programming. However, if an agile approach to development is used, design and implementation are interleaved, with no formal design documents produced during the process. Of course, the software is still designed, but the design is recorded informally on whiteboards and programmer's notebooks.

A software design is a description of the structure of the software to be implemented, the data models and structures used by the system, the interfaces between system components and, sometimes, the algorithms used. Designers do not arrive at a finished design immediately but develop the design in stages. They add detail as they develop their design, with constant backtracking to modify earlier designs.

Figure 2.5 is an abstract model of the design process showing the inputs to the design process, process activities, and the process outputs. The design process activities are both interleaved and interdependent. New information about the design is constantly being generated, and this affects previous design decisions. Design rework is therefore inevitable.

Most software interfaces with other software systems. These other systems include the operating system, database, middleware, and other application systems. These make up the “software platform,” the environment in which the software will execute. Information about this platform is an essential input to the design process, as designers must decide how best to integrate it with its environment. If the system is to process existing data, then the description of that data may be included in the platform specification. Otherwise, the data description must be an input to the design process so that the system data organization can be defined.

The activities in the design process vary, depending on the type of system being developed. For example, real-time systems require an additional stage of timing design but may not include a database, so there is no database design involved. Figure 2.5 shows four activities that may be part of the design process for information systems:

1. *Architectural design*, where you identify the overall structure of the system, the principal components (sometimes called subsystems or modules), their relationships, and how they are distributed.
2. *Database design*, where you design the system data structures and how these are to be represented in a database. Again, the work here depends on whether an existing database is to be reused or a new database is to be created.
3. *Interface design*, where you define the interfaces between system components. This interface specification must be unambiguous. With a precise interface, a component may be used by other components without them having to know how it is implemented. Once interface specifications are agreed, the components can be separately designed and developed.
4. *Component selection and design*, where you search for reusable components and, if no suitable components are available, design new software components. The design at this stage may be a simple component description with the implementation details left to the programmer. Alternatively, it may be a list of changes to be made to a reusable component or a detailed design model expressed in the UML. The design model may then be used to automatically generate an implementation.

These activities lead to the design outputs, which are also shown in Figure 2.5. For critical systems, the outputs of the design process are detailed design documents setting out precise and accurate descriptions of the system. If a model-driven approach is used (Chapter 5), the design outputs are design diagrams. Where agile methods of development are used, the outputs of the design process may not be separate specification documents but may be represented in the code of the program.

The development of a program to implement a system follows naturally from system design. Although some classes of program, such as safety-critical systems, are usually designed in detail before any implementation begins, it is more common for design and program development to be interleaved. Software development tools may be used to generate a skeleton program from a design. This includes code to

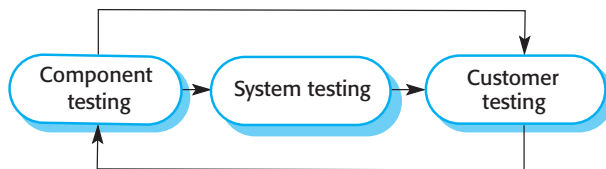


Figure 2.6 Stages of testing

define and implement interfaces, and, in many cases, the developer need only add details of the operation of each program component.

Programming is an individual activity, and there is no general process that is usually followed. Some programmers start with components that they understand, develop these, and then move on to less understood components. Others take the opposite approach, leaving familiar components till last because they know how to develop them. Some developers like to define data early in the process and then use this to drive the program development; others leave data unspecified for as long as possible.

Normally, programmers carry out some testing of the code they have developed. This often reveals program defects (bugs) that must be removed from the program. Finding and fixing program defects is called *debugging*. Defect testing and debugging are different processes. Testing establishes the existence of defects. Debugging is concerned with locating and correcting these defects.

When you are debugging, you have to generate hypotheses about the observable behavior of the program and then test these hypotheses in the hope of finding the fault that caused the output anomaly. Testing the hypotheses may involve tracing the program code manually. It may require new test cases to localize the problem. Interactive debugging tools, which show the intermediate values of program variables and a trace of the statements executed, are usually used to support the debugging process.

2.2.3 Software validation

Software validation or, more generally, verification and validation (V & V) is intended to show that a system both conforms to its specification and meets the expectations of the system customer. Program testing, where the system is executed using simulated test data, is the principal validation technique. Validation may also involve checking processes, such as inspections and reviews, at each stage of the software process from user requirements definition to program development. However, most V & V time and effort is spent on program testing.

Except for small programs, systems should not be tested as a single, monolithic unit. Figure 2.6 shows a three-stage testing process in which system components are individually tested, then the integrated system is tested. For custom software, customer testing involves testing the system with real customer data. For products that are sold as applications, customer testing is sometimes called beta testing where selected users try out and comment on the software.

The stages in the testing process are:

1. *Component testing* The components making up the system are tested by the people developing the system. Each component is tested independently, without other system components. Components may be simple entities such as functions or object classes or may be coherent groupings of these entities. Test automation tools, such as JUnit for Java, that can rerun tests when new versions of the component are created, are commonly used (Koskela 2013).
2. *System testing* System components are integrated to create a complete system. This process is concerned with finding errors that result from unanticipated interactions between components and component interface problems. It is also concerned with showing that the system meets its functional and non-functional requirements, and testing the emergent system properties. For large systems, this may be a multistage process where components are integrated to form subsystems that are individually tested before these subsystems are integrated to form the final system.
3. *Customer testing* This is the final stage in the testing process before the system is accepted for operational use. The system is tested by the system customer (or potential customer) rather than with simulated test data. For custom-built software, customer testing may reveal errors and omissions in the system requirements definition, because the real data exercise the system in different ways from the test data. Customer testing may also reveal requirements problems where the system's facilities do not really meet the users' needs or the system performance is unacceptable. For products, customer testing shows how well the software product meets the customer's needs.

Ideally, component defects are discovered early in the testing process, and interface problems are found when the system is integrated. However, as defects are discovered, the program must be debugged, and this may require other stages in the testing process to be repeated. Errors in program components, say, may come to light during system testing. The process is therefore an iterative one with information being fed back from later stages to earlier parts of the process.

Normally, component testing is simply part of the normal development process. Programmers make up their own test data and incrementally test the code as it is developed. The programmer knows the component and is therefore the best person to generate test cases.

If an incremental approach to development is used, each increment should be tested as it is developed, with these tests based on the requirements for that increment. In test-driven development, which is a normal part of agile processes, tests are developed along with the requirements before development starts. This helps the testers and developers to understand the requirements and ensures that there are no delays as test cases are created.

When a plan-driven software process is used (e.g., for critical systems development), testing is driven by a set of test plans. An independent team of testers works

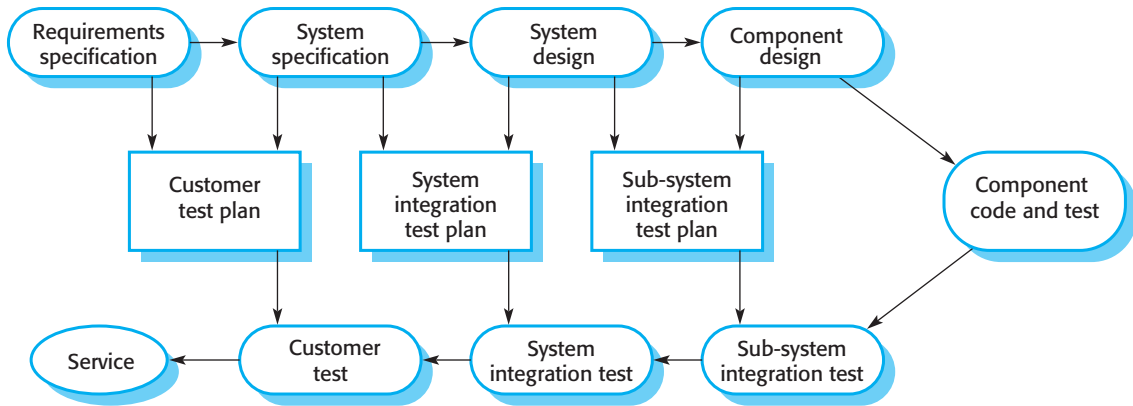


Figure 2.7 Testing phases in a plan-driven software process

from these test plans, which have been developed from the system specification and design. Figure 2.7 illustrates how test plans are the link between testing and development activities. This is sometimes called the V-model of development (turn it on its side to see the V). The V-model shows the software validation activities that correspond to each stage of the waterfall process model.

When a system is to be marketed as a software product, a testing process called beta testing is often used. Beta testing involves delivering a system to a number of potential customers who agree to use that system. They report problems to the system developers. This exposes the product to real use and detects errors that may not have been anticipated by the product developers. After this feedback, the software product may be modified and released for further beta testing or general sale.

2.2.4 Software evolution

The flexibility of software is one of the main reasons why more and more software is being incorporated into large, complex systems. Once a decision has been made to manufacture hardware, it is very expensive to make changes to the hardware design. However, changes can be made to software at any time during or after the system development. Even extensive changes are still much cheaper than corresponding changes to system hardware.

Historically, there has always been a split between the process of software development and the process of software evolution (software maintenance). People think of software development as a creative activity in which a software system is developed from an initial concept through to a working system. However, they sometimes think of software maintenance as dull and uninteresting. They think that software maintenance is less interesting and challenging than original software development.

This distinction between development and maintenance is increasingly irrelevant. Very few software systems are completely new systems, and it makes much more

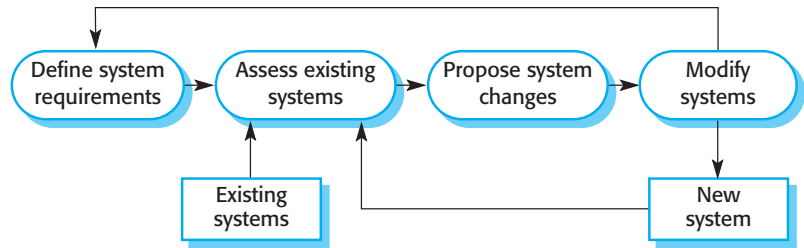


Figure 2.8 Software system evolution

sense to see development and maintenance as a continuum. Rather than two separate processes, it is more realistic to think of software engineering as an evolutionary process (Figure 2.8) where software is continually changed over its lifetime in response to changing requirements and customer needs.

2.3 Coping with change

Change is inevitable in all large software projects. The system requirements change as businesses respond to external pressures, competition, and changed management priorities. As new technologies become available, new approaches to design and implementation become possible. Therefore whatever software process model is used, it is essential that it can accommodate changes to the software being developed.

Change adds to the costs of software development because it usually means that work that has been completed has to be redone. This is called rework. For example, if the relationships between the requirements in a system have been analyzed and new requirements are then identified, some or all of the requirements analysis has to be repeated. It may then be necessary to redesign the system to deliver the new requirements, change any programs that have been developed, and retest the system.

Two related approaches may be used to reduce the costs of rework:

1. *Change anticipation*, where the software process includes activities that can anticipate or predict possible changes before significant rework is required. For example, a prototype system may be developed to show some key features of the system to customers. They can experiment with the prototype and refine their requirements before committing to high software production costs.
2. *Change tolerance*, where the process and software are designed so that changes can be easily made to the system. This normally involves some form of incremental development. Proposed changes may be implemented in increments that have not yet been developed. If this is impossible, then only a single increment (a small part of the system) may have to be altered to incorporate the change.

In this section, I discuss two ways of coping with change and changing system requirements:

1. *System prototyping*, where a version of the system or part of the system is developed quickly to check the customer's requirements and the feasibility of design decisions. This is a method of change anticipation as it allows users to experiment with the system before delivery and so refine their requirements. The number of requirements change proposals made after delivery is therefore likely to be reduced.
2. *Incremental delivery*, where system increments are delivered to the customer for comment and experimentation. This supports both change avoidance and change tolerance. It avoids the premature commitment to requirements for the whole system and allows changes to be incorporated into later increments at relatively low cost.

The notion of refactoring, namely, improving the structure and organization of a program, is also an important mechanism that supports change tolerance. I discuss this in Chapter 3 (Agile methods).

2.3.1 Prototyping

A prototype is an early version of a software system that is used to demonstrate concepts, try out design options, and find out more about the problem and its possible solutions. Rapid, iterative development of the prototype is essential so that costs are controlled and system stakeholders can experiment with the prototype early in the software process.

A software prototype can be used in a software development process to help anticipate changes that may be required:

1. In the requirements engineering process, a prototype can help with the elicitation and validation of system requirements.
2. In the system design process, a prototype can be used to explore software solutions and in the development of a user interface for the system.

System prototypes allow potential users to see how well the system supports their work. They may get new ideas for requirements and find areas of strength and weakness in the software. They may then propose new system requirements. Furthermore, as the prototype is developed, it may reveal errors and omissions in the system requirements. A feature described in a specification may seem to be clear and useful. However, when that function is combined with other functions, users often find that their initial view was incorrect or incomplete. The system specification can then be modified to reflect the changed understanding of the requirements.

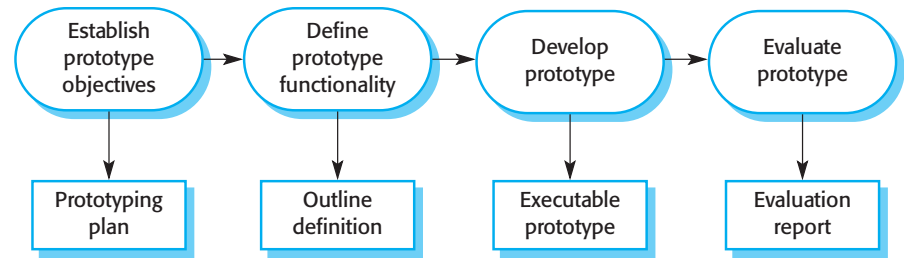


Figure 2.9 Prototype development

A system prototype may be used while the system is being designed to carry out design experiments to check the feasibility of a proposed design. For example, a database design may be prototyped and tested to check that it supports efficient data access for the most common user queries. Rapid prototyping with end-user involvement is the only sensible way to develop user interfaces. Because of the dynamic nature of user interfaces, textual descriptions and diagrams are not good enough for expressing the user interface requirements and design.

A process model for prototype development is shown in Figure 2.9. The objectives of prototyping should be made explicit from the start of the process. These may be to develop the user interface, to develop a system to validate functional system requirements, or to develop a system to demonstrate the application to managers. The same prototype usually cannot meet all objectives. If the objectives are left unstated, management or end-users may misunderstand the function of the prototype. Consequently, they may not get the benefits that they expected from the prototype development.

The next stage in the process is to decide what to put into and, perhaps more importantly, what to leave out of the prototype system. To reduce prototyping costs and accelerate the delivery schedule, you may leave some functionality out of the prototype. You may decide to relax non-functional requirements such as response time and memory utilization. Error handling and management may be ignored unless the objective of the prototype is to establish a user interface. Standards of reliability and program quality may be reduced.

The final stage of the process is prototype evaluation. Provision must be made during this stage for user training, and the prototype objectives should be used to derive a plan for evaluation. Potential users need time to become comfortable with a new system and to settle into a normal pattern of usage. Once they are using the system normally, they then discover requirements errors and omissions. A general problem with prototyping is that users may not use the prototype in the same way as they use the final system. Prototype testers may not be typical of system users. There may not be enough time to train users during prototype evaluation. If the prototype is slow, the evaluators may adjust their way of working and avoid those system features that have slow response times. When provided with better response in the final system, they may use it in a different way.

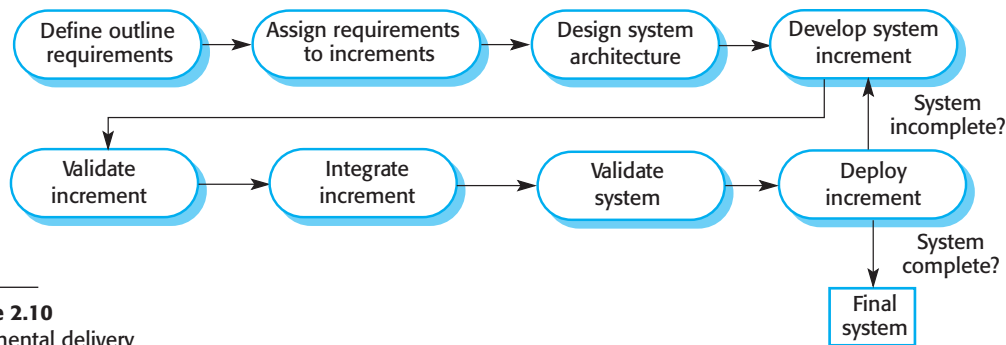


Figure 2.10
Incremental delivery

2.3.2 Incremental delivery

Incremental delivery (Figure 2.10) is an approach to software development where some of the developed increments are delivered to the customer and deployed for use in their working environment. In an incremental delivery process, customers define which of the services are most important and which are least important to them. A number of delivery increments are then defined, with each increment providing a subset of the system functionality. The allocation of services to increments depends on the service priority, with the highest priority services implemented and delivered first.

Once the system increments have been identified, the requirements for the services to be delivered in the first increment are defined in detail and that increment is developed. During development, further requirements analysis for later increments can take place, but requirements changes for the current increment are not accepted.

Once an increment is completed and delivered, it is installed in the customer's normal working environment. They can experiment with the system, and this helps them clarify their requirements for later system increments. As new increments are completed, they are integrated with existing increments so that system functionality improves with each delivered increment.

Incremental delivery has a number of advantages:

1. Customers can use the early increments as prototypes and gain experience that informs their requirements for later system increments. Unlike prototypes, these are part of the real system, so there is no relearning when the complete system is available.
2. Customers do not have to wait until the entire system is delivered before they can gain value from it. The first increment satisfies their most critical requirements, so they can use the software immediately.
3. The process maintains the benefits of incremental development in that it should be relatively easy to incorporate changes into the system.

4. As the highest priority services are delivered first and later increments then integrated, the most important system services receive the most testing. This means that customers are less likely to encounter software failures in the most important parts of the system.

However, there are problems with incremental delivery. In practice, it only works in situations where a brand-new system is being introduced and the system evaluators are given time to experiment with the new system. Key problems with this approach are:

1. Iterative delivery is problematic when the new system is intended to replace an existing system. Users need all of the functionality of the old system and are usually unwilling to experiment with an incomplete new system. It is often impractical to use the old and the new systems alongside each other as they are likely to have different databases and user interfaces.
2. Most systems require a set of basic facilities that are used by different parts of the system. As requirements are not defined in detail until an increment is to be implemented, it can be hard to identify common facilities that are needed by all increments.
3. The essence of iterative processes is that the specification is developed in conjunction with the software. However, this conflicts with the procurement model of many organizations, where the complete system specification is part of the system development contract. In the incremental approach, there is no complete system specification until the final increment is specified. This requires a new form of contract, which large customers such as government agencies may find difficult to accommodate.

For some types of systems, incremental development and delivery is not the best approach. These are very large systems where development may involve teams working in different locations, some embedded systems where the software depends on hardware development, and some critical systems where all the requirements must be analyzed to check for interactions that may compromise the safety or security of the system.

These large systems, of course, suffer from the same problems of uncertain and changing requirements. Therefore, to address these problems and get some of the benefits of incremental development, a system prototype may be developed and used as a platform for experiments with the system requirements and design. With the experience gained from the prototype, definitive requirements can then be agreed.

2.4 Process improvement

Nowadays, there is a constant demand from industry for cheaper, better software, which has to be delivered to ever-tighter deadlines. Consequently, many software companies have turned to software process improvement as a way of enhancing the

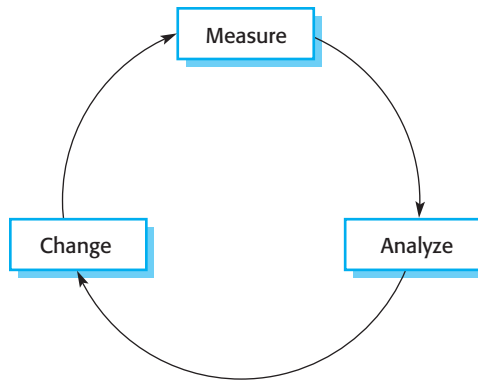


Figure 2.11 The process improvement cycle

quality of their software, reducing costs, or accelerating their development processes. Process improvement means understanding existing processes and changing these processes to increase product quality and/or reduce costs and development time. I cover general issues of process measurement and process improvement in detail in web Chapter 26.

Two quite different approaches to process improvement and change are used:

1. The process maturity approach, which has focused on improving process and project management and introducing good software engineering practice into an organization. The level of process maturity reflects the extent to which good technical and management practice has been adopted in organizational software development processes. The primary goals of this approach are improved product quality and process predictability.
2. The agile approach, which has focused on iterative development and the reduction of overheads in the software process. The primary characteristics of agile methods are rapid delivery of functionality and responsiveness to changing customer requirements. The improvement philosophy here is that the best processes are those with the lowest overheads and agile approaches can achieve this. I describe agile approaches in Chapter 3.

People who are enthusiastic about and committed to each of these approaches are generally skeptical of the benefits of the other. The process maturity approach is rooted in plan-driven development and usually requires increased “overhead,” in the sense that activities are introduced that are not directly relevant to program development. Agile approaches focus on the code being developed and deliberately minimize formality and documentation.

The general process improvement process underlying the process maturity approach is a cyclical process, as shown in Figure 2.11. The stages in this process are:

1. *Process measurement* You measure one or more attributes of the software process or product. These measurements form a baseline that helps you decide if

process improvements have been effective. As you introduce improvements, you re-measure the same attributes, which will hopefully have improved in some way.

2. *Process analysis* The current process is assessed, and process weaknesses and bottlenecks are identified. Process models (sometimes called process maps) that describe the process may be developed during this stage. The analysis may be focused by considering process characteristics such as rapidity and robustness.
3. *Process change* Process changes are proposed to address some of the identified process weaknesses. These are introduced, and the cycle resumes to collect data about the effectiveness of the changes.

Without concrete data on a process or the software developed using that process, it is impossible to assess the value of process improvement. However, companies starting the process improvement process are unlikely to have process data available as an improvement baseline. Therefore, as part of the first cycle of changes, you may have to collect data about the software process and to measure software product characteristics.

Process improvement is a long-term activity, so each of the stages in the improvement process may last several months. It is also a continuous activity as, whatever new processes are introduced, the business environment will change and the new processes will themselves have to evolve to take these changes into account.

The notion of process maturity was introduced in the late 1980s when the Software Engineering Institute (SEI) proposed their model of process capability maturity (Humphrey 1988). The maturity of a software company's processes reflects the process management, measurement, and use of good software engineering practices in the company. This idea was introduced so that the U.S. Department of Defense could assess the software engineering capability of defense contractors, with a view to limiting contracts to those contractors who had reached a required level of process maturity. Five levels of process maturity were proposed, as shown in Figure 2.12. These have evolved and developed over the last 25 years (Chrissis, Konrad, and Shrum 2011), but the fundamental ideas in Humphrey's model are still the basis of software process maturity assessment.

The levels in the process maturity model are:

1. *Initial* The goals associated with the process area are satisfied, and for all processes the scope of the work to be performed is explicitly set out and communicated to the team members.
2. *Managed* At this level, the goals associated with the process area are met, and organizational policies are in place that define when each process should be used. There must be documented project plans that define the project goals. Resource management and process monitoring procedures must be in place across the institution.
3. *Defined* This level focuses on organizational standardization and deployment of processes. Each project has a managed process that is adapted to the project requirements from a defined set of organizational processes. Process assets and process measurements must be collected and used for future process improvements.

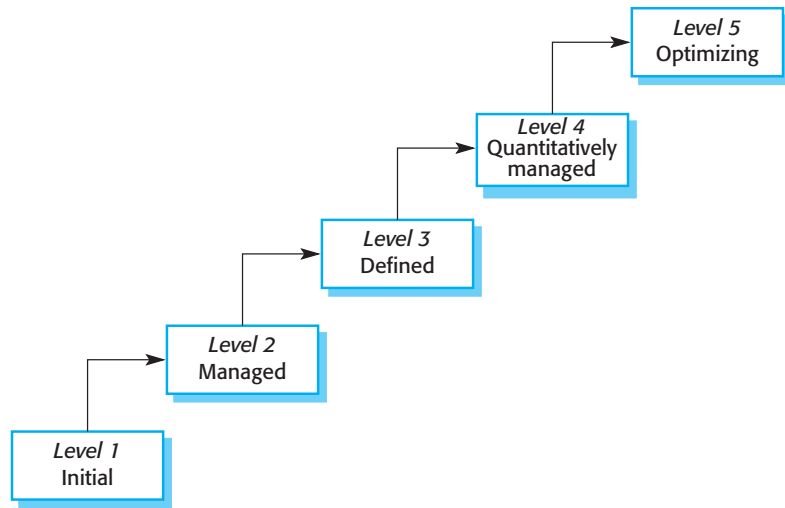


Figure 2.12 Capability maturity levels

4. *Quantitatively managed* At this level, there is an organizational responsibility to use statistical and other quantitative methods to control subprocesses. That is, collected process and product measurements must be used in process management.
5. *Optimizing* At this highest level, the organization must use the process and product measurements to drive process improvement. Trends must be analyzed and the processes adapted to changing business needs.

The work on process maturity levels has had a major impact on the software industry. It focused attention on the software engineering processes and practices that were used and led to significant improvements in software engineering capability. However, there is too much overhead in formal process improvement for small companies, and maturity estimation with agile processes is difficult. Consequently, only large software companies now use this maturity-focused approach to software process improvement.

KEY POINTS

- Software processes are the activities involved in producing a software system. Software process models are abstract representations of these processes.
- General process models describe the organization of software processes. Examples of these general models include the waterfall model, incremental development, and reusable component configuration and integration.

- Requirements engineering is the process of developing a software specification. Specifications are intended to communicate the system needs of the customer to the system developers.
- Design and implementation processes are concerned with transforming a requirements specification into an executable software system.
- Software validation is the process of checking that the system conforms to its specification and that it meets the real needs of the users of the system.
- Software evolution takes place when you change existing software systems to meet new requirements. Changes are continuous, and the software must evolve to remain useful.
- Processes should include activities to cope with change. This may involve a prototyping phase that helps avoid poor decisions on requirements and design. Processes may be structured for iterative development and delivery so that changes may be made without disrupting the system as a whole.
- Process improvement is the process of improving existing software processes to improve software quality, lower development costs, or reduce development time. It is a cyclic process involving process measurement, analysis, and change.

FURTHER READING

“Process Models in Software Engineering.” This is an excellent overview of a wide range of software engineering process models that have been proposed. (W. Scacchi, *Encyclopaedia of Software Engineering*, ed. J. J. Marciniak, John Wiley & Sons, 2001) <http://www.ics.uci.edu/~wscacchi/Papers/SE-Encyc/Process-Models-SE-Encyc.pdf>

Software Process Improvement: Results and Experience from the Field. This book is a collection of papers focusing on process improvement case studies in several small and medium-sized Norwegian companies. It also includes a good introduction to the general issues of process improvement. (Conradi, R., Dybå, T., Sjøberg, D., and Ulsund, T. (eds.), Springer, 2006).

“Software Development Life Cycle Models and Methodologies.” This blog post is a succinct summary of several software process models that have been proposed and used. It discusses the advantages and disadvantages of each of these models (M. Sami, 2012). <http://melsatar.wordpress.com/2012/03/15/software-development-life-cycle-models-and-methodologies/>

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

<http://software-engineering-book.com/videos/software-engineering/>

EXERCISES

- 2.1.** Suggest the most appropriate generic software process model that might be used as a basis for managing the development of the following systems. Explain your answer according to the type of system being developed:

A system to control antilock braking in a car

A virtual reality system to support software maintenance

A university accounting system that replaces an existing system

An interactive travel planning system that helps users plan journeys with the lowest environmental impact

- 2.2.** Incremental software development could be very effectively used for customers who do not have a clear idea about the systems needed for their operations. Discuss.
- 2.3.** Consider the integration and configuration process model shown in Figure 2.3. Explain why it is essential to repeat the requirements engineering activity in the process.
- 2.4.** Suggest why it is important to make a distinction between developing the user requirements and developing system requirements in the requirements engineering process.
- 2.5.** Using an example, explain why the design activities of architectural design, database design, interface design, and component design are interdependent.
- 2.6.** Explain why software testing should always be an incremental, staged activity. Are programmers the best people to test the programs that they have developed?
- 2.7.** Imagine that a government wants a software program that helps to keep track of the utilization of the country's vast mineral resources. Although the requirements put forward by the government were not very clear, a software company was tasked with the development of a prototype. The government found the prototype impressive, and asked it be extended to be the actual system that would be used. Discuss the pros and cons of taking this approach.
- 2.8.** You have developed a prototype of a software system and your manager is very impressed by it. She proposes that it should be put into use as a production system, with new features added as required. This avoids the expense of system development and makes the system immediately useful. Write a short report for your manager explaining why prototype systems should not normally be used as production systems.
- 2.9.** Suggest two advantages and two disadvantages of the approach to process assessment and improvement that is embodied in the SEI's Capability Maturity framework.
- 2.10.** Historically, the introduction of technology has caused profound changes in the labor market and, temporarily at least, displaced people from jobs. Discuss whether the introduction of extensive process automation is likely to have the same consequences for software engineers. If you don't think it will, explain why not. If you think that it will reduce job opportunities, is it ethical for the engineers affected to passively or actively resist the introduction of this technology?

REFERENCES

- Abrial, J. R. 2005. *The B Book: Assigning Programs to Meanings*. Cambridge, UK: Cambridge University Press.
- . 2010. *Modeling in Event-B: System and Software Engineering*. Cambridge, UK: Cambridge University Press.
- Boehm, B. W. (1988). “A Spiral Model of Software Development and Enhancement.” *IEEE Computer*, 21 (5), 61–72. doi:10.1145/12944.12948
- Boehm, B. W., and R. Turner. 2004. “Balancing Agility and Discipline: Evaluating and Integrating Agile and Plan-Driven Methods.” In *26th Int. Conf on Software Engineering*, Edinburgh, Scotland. doi:10.1109/ICSE.2004.1317503.
- Chrissis, M. B., M. Konrad, and S. Shrum. 2011. *CMMI for Development: Guidelines for Process Integration and Product Improvement, 3rd ed.* Boston: Addison-Wesley.
- Humphrey, W. S. 1988. “Characterizing the Software Process: A Maturity Framework.” *IEEE Software* 5 (2): 73–79. doi:10.1109/2.59.
- Koskela, L. 2013. *Effective Unit Testing: A Guide for Java Developers*. Greenwich, CT: Manning Publications.
- Krutchen, P. 2003. *The Rational Unified Process—An Introduction, 3rd ed.* Reading, MA: Addison-Wesley.
- Royce, W. W. 1970. “Managing the Development of Large Software Systems: Concepts and Techniques.” In *IEEE WESTCON*, 1–9. Los Angeles, CA.
- Wheeler, W., and J. White. 2013. *Spring in Practice*. Greenwich, CT: Manning Publications.



3

Agile software development

Objectives

The objective of this chapter is to introduce you to agile software development methods. When you have read the chapter, you will:

- understand the rationale for agile software development methods, the agile manifesto, and the differences between agile and plan-driven development;
- know about important agile development practices such as user stories, refactoring, pair programming and test-first development;
- understand the Scrum approach to agile project management;
- understand the issues of scaling agile development methods and combining agile approaches with plan-driven approaches in the development of large software systems.

Contents

- 3.1** Agile methods
- 3.2** Agile development techniques
- 3.3** Agile project management
- 3.4** Scaling agile methods

Businesses now operate in a global, rapidly changing environment. They have to respond to new opportunities and markets, changing economic conditions and the emergence of competing products and services. Software is part of almost all business operations, so new software has to be developed quickly to take advantage of new opportunities and to respond to competitive pressure. Rapid software development and delivery is therefore the most critical requirement for most business systems. In fact, businesses may be willing to trade off software quality and compromise on requirements if they can deploy essential new software quickly.

Because these businesses are operating in a changing environment, it is practically impossible to derive a complete set of stable software requirements. Requirements change because customers find it impossible to predict how a system will affect working practices, how it will interact with other systems, and what user operations should be automated. It may only be after a system has been delivered and users gain experience with it that the real requirements become clear. Even then, external factors drive requirements change.

Plan-driven software development processes that completely specify the requirements and then design, build, and test a system are not geared to rapid software development. As the requirements change or as requirements problems are discovered, the system design or implementation has to be reworked and retested. As a consequence, a conventional waterfall or specification-based process is usually a lengthy one, and the final software is delivered to the customer long after it was originally specified.

For some types of software, such as safety-critical control systems, where a complete analysis of the system is essential, this plan-driven approach is the right one. However, in a fast-moving business environment, it can cause real problems. By the time the software is available for use, the original reason for its procurement may have changed so radically that the software is effectively useless. Therefore, for business systems in particular, development processes that focus on rapid software development and delivery are essential.

The need for rapid software development and processes that can handle changing requirements has been recognized for many years (Larman and Basili 2003). However, faster software development really took off in the late 1990s with the development of the idea of “agile methods” such as Extreme Programming (Beck 1999), Scrum (Schwaber and Beedle 2001), and DSDM (Stapleton 2003).

Rapid software development became known as agile development or agile methods. These agile methods are designed to produce useful software quickly. All of the agile methods that have been proposed share a number of common characteristics:

1. The processes of specification, design and implementation are interleaved. There is no detailed system specification, and design documentation is minimized or generated automatically by the programming environment used to implement the system. The user requirements document is an outline definition of the most important characteristics of the system.
2. The system is developed in a series of increments. End-users and other system stakeholders are involved in specifying and evaluating each increment.

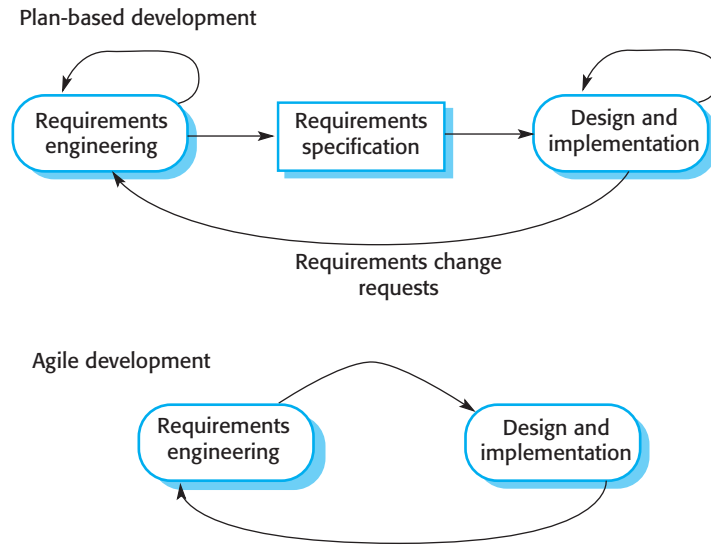


Figure 3.1 Plan-driven and agile development

They may propose changes to the software and new requirements that should be implemented in a later version of the system.

3. Extensive tool support is used to support the development process. Tools that may be used include automated testing tools, tools to support configuration management, and system integration and tools to automate user interface production.

Agile methods are incremental development methods in which the increments are small, and, typically, new releases of the system are created and made available to customers every two or three weeks. They involve customers in the development process to get rapid feedback on changing requirements. They minimize documentation by using informal communications rather than formal meetings with written documents.

Agile approaches to software development consider design and implementation to be the central activities in the software process. They incorporate other activities, such as requirements elicitation and testing, into design and implementation. By contrast, a plan-driven approach to software engineering identifies separate stages in the software process with outputs associated with each stage. The outputs from one stage are used as a basis for planning the following process activity.

Figure 3.1 shows the essential distinctions between plan-driven and agile approaches to system specification. In a plan-driven software development process, iteration occurs within activities, with formal documents used to communicate between stages of the process. For example, the requirements will evolve, and, ultimately, a requirements specification will be produced. This is then an input to the design and implementation process. In an agile approach, iteration occurs across activities. Therefore, the requirements and the design are developed together rather than separately.

In practice, as I explain in Section 3.4.1, plan-driven processes are often used along with agile programming practices, and agile methods may incorporate some planned

activities apart from programming and testing. It is perfectly feasible, in a plan-driven process, to allocate requirements and plan the design and development phase as a series of increments. An agile process is not inevitably code-focused, and it may produce some design documentation. Agile developers may decide that an iteration should not produce new code but rather should produce system models and documentation.

3.1 Agile methods

In the 1980s and early 1990s, there was a widespread view that the best way to achieve better software was through careful project planning, formalized quality assurance, use of analysis and design methods supported by software tools, and controlled and rigorous software development processes. This view came from the software engineering community that was responsible for developing large, long-lived software systems such as aerospace and government systems.

This plan-driven approach was developed for software developed by large teams, working for different companies. Teams were often geographically dispersed and worked on the software for long periods of time. An example of this type of software is the control systems for a modern aircraft, which might take up to 10 years from initial specification to deployment. Plan-driven approaches involve a significant overhead in planning, designing, and documenting the system. This overhead is justified when the work of multiple development teams has to be coordinated, when the system is a critical system, and when many different people will be involved in maintaining the software over its lifetime.

However, when this heavyweight, plan-driven development approach is applied to small and medium-sized business systems, the overhead involved is so large that it dominates the software development process. More time is spent on how the system should be developed than on program development and testing. As the system requirements change, rework is essential and, in principle at least, the specification and design have to change with the program.

Dissatisfaction with these heavyweight approaches to software engineering led to the development of agile methods in the late 1990s. These methods allowed the development team to focus on the software itself rather than on its design and documentation. They are best suited to application development where the system requirements usually change rapidly during the development process. They are intended to deliver working software quickly to customers, who can then propose new and changed requirements to be included in later iterations of the system. They aim to cut down on process bureaucracy by avoiding work that has dubious long-term value and eliminating documentation that will probably never be used.

The philosophy behind agile methods is reflected in the agile manifesto (<http://agilemanifesto.org>) issued by the leading developers of these methods. This manifesto states:

Principle	Description
Customer involvement	Customers should be closely involved throughout the development process. Their role is provide and prioritize new system requirements and to evaluate the iterations of the system.
Embrace change	Expect the system requirements to change, and so design the system to accommodate these changes.
Incremental delivery	The software is developed in increments, with the customer specifying the requirements to be included in each increment.
Maintain simplicity	Focus on simplicity in both the software being developed and in the development process. Wherever possible, actively work to eliminate complexity from the system.
People, not process	The skills of the development team should be recognized and exploited. Team members should be left to develop their own ways of working without prescriptive processes.

Figure 3.2 The principles of agile methods

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

Individuals and interactions over processes and tools

Working software over comprehensive documentation

Customer collaboration over contract negotiation

Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more[†].

All agile methods suggest that software should be developed and delivered incrementally. These methods are based on different agile processes but they share a set of principles, based on the agile manifesto, and so they have much in common. I have listed these principles in Figure 3.2.

Agile methods have been particularly successful for two kinds of system development.

1. Product development where a software company is developing a small or medium-sized product for sale. Virtually all software products and apps are now developed using an agile approach.
2. Custom system development within an organization, where there is a clear commitment from the customer to become involved in the development process and where there are few external stakeholders and regulations that affect the software.

Agile methods work well in these situations because it is possible to have continuous communications between the product manager or system customer and the development team. The software itself is a stand-alone system rather than tightly integrated with other systems being developed at the same time. Consequently, there is no need to coordinate parallel development streams. Small and medium-sized

[†]<http://agilemanifesto.org/>

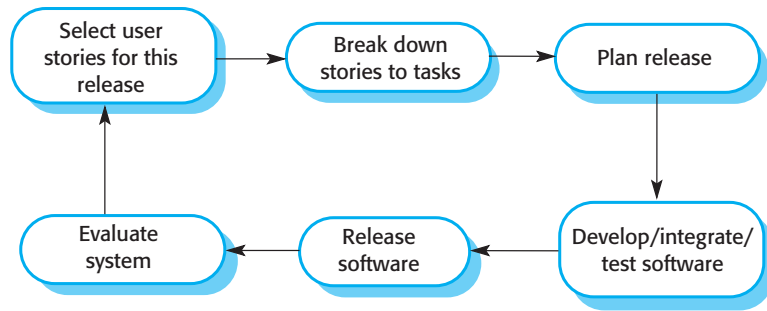


Figure 3.3 The XP release cycle

systems can be developed by co-located teams, so informal communications among team members work well.

3.2 Agile development techniques

The ideas underlying agile methods were developed around the same time by a number of different people in the 1990s. However, perhaps the most significant approach to changing software development culture was the development of Extreme Programming (XP). The name was coined by Kent Beck (Beck 1998) because the approach was developed by pushing recognized good practice, such as iterative development, to “extreme” levels. For example, in XP, several new versions of a system may be developed by different programmers, integrated, and tested in a day. Figure 3.3 illustrates the XP process to produce an increment of the system that is being developed.

In XP, requirements are expressed as scenarios (called user stories), which are implemented directly as a series of tasks. Programmers work in pairs and develop tests for each task before writing the code. All tests must be successfully executed when new code is integrated into the system. There is a short time gap between releases of the system.

Extreme programming was controversial as it introduced a number of agile practices that were quite different from the development practice of that time. These practices are summarized in Figure 3.4 and reflect the principles of the agile manifesto:

1. Incremental development is supported through small, frequent releases of the system. Requirements are based on simple customer stories or scenarios that are used as a basis for deciding what functionality should be included in a system increment.
2. Customer involvement is supported through the continuous engagement of the customer in the development team. The customer representative takes part in the development and is responsible for defining acceptance tests for the system.
3. People, not process, are supported through pair programming, collective ownership of the system code, and a sustainable development process that does not involve excessively long working hours.

Principle or practice	Description
Collective ownership	The pairs of developers work on all areas of the system, so that no islands of expertise develop and all the developers take responsibility for all of the code. Anyone can change anything.
Continuous integration	As soon as the work on a task is complete, it is integrated into the whole system. After any such integration, all the unit tests in the system must pass.
Incremental planning	Requirements are recorded on “story cards,” and the stories to be included in a release are determined by the time available and their relative priority. The developers break these stories into development “tasks.” See Figures 3.5 and 3.6.
On-site customer	A representative of the end-user of the system (the Customer) should be available full time for the use of the XP team. In an extreme programming process, the customer is a member of the development team and is responsible for bringing system requirements to the team for implementation.
Pair programming	Developers work in pairs, checking each other's work and providing the support to always do a good job.
Refactoring	All developers are expected to refactor the code continuously as soon as potential code improvements are found. This keeps the code simple and maintainable.
Simple design	Enough design is carried out to meet the current requirements and no more.
Small releases	The minimal useful set of functionality that provides business value is developed first. Releases of the system are frequent and incrementally add functionality to the first release.
Sustainable pace	Large amounts of overtime are not considered acceptable, as the net effect is often to reduce code quality and medium-term productivity.
Test first development	An automated unit test framework is used to write tests for a new piece of functionality before that functionality itself is implemented.

Figure 3.4 Extreme programming practices

4. Change is embraced through regular system releases to customers, test-first development, refactoring to avoid code degeneration, and continuous integration of new functionality.
5. Maintaining simplicity is supported by constant refactoring that improves code quality and by using simple designs that do not unnecessarily anticipate future changes to the system.

In practice, the application of Extreme Programming as originally proposed has proved to be more difficult than anticipated. It cannot be readily integrated with the management practices and culture of most businesses. Therefore, companies adopting agile methods pick and choose those XP practices that are most appropriate for their way of working. Sometimes these are incorporated into their own development processes but, more commonly, they are used in conjunction with a management-focused agile method such as Scrum (Rubin 2013).

Prescribing medication

Kate is a doctor who wishes to prescribe medication for a patient attending a clinic. The patient record is already displayed on her computer so she clicks on the medication field and can select 'current medication', 'new medication' or 'formulary'.

If she selects 'current medication', the system asks her to check the dose; if she wants to change the dose, she enters the new dose then confirms the prescription.

If she chooses 'new medication', the system assumes that she knows which medication to prescribe. She types the first few letters of the drug name. The system displays a list of possible drugs starting with these letters. She chooses the required medication and the system responds by asking her to check that the medication selected is correct. She enters the dose then confirms the prescription.

If she chooses 'formulary', the system displays a search box for the approved formulary. She can then search for the drug required. She selects a drug and is asked to check that the medication is correct. She enters the dose then confirms the prescription.

The system always checks that the dose is within the approved range. If it isn't, Kate is asked to change the dose.

After Kate has confirmed the prescription, it will be displayed for checking. She either clicks 'OK' or 'Change'. If she clicks 'OK', the prescription is recorded on the audit database. If she clicks on 'Change', she reenters the 'Prescribing medication' process.

Figure 3.5 A
"prescribing medication"
story

I am not convinced that XP on its own is a practical agile method for most companies, but its most significant contribution is probably the set of agile development practices that it introduced to the community. I discuss the most important of these practices in this section.

3.2.1 User stories

Software requirements always change. To handle these changes, agile methods do not have a separate requirements engineering activity. Rather, they integrate requirements elicitation with development. To make this easier, the idea of "user stories" was developed where a user story is a scenario of use that might be experienced by a system user.

As far as possible, the system customer works closely with the development team and discusses these scenarios with other team members. Together, they develop a "story card" that briefly describes a story that encapsulates the customer needs. The development team then aims to implement that scenario in a future release of the software. An example of a story card for the Mentcare system is shown in Figure 3.5. This is a short description of a scenario for prescribing medication for a patient.

User stories may be used in planning system iterations. Once the story cards have been developed, the development team breaks these down into tasks (Figure 3.6) and estimates the effort and resources required for implementing each task. This usually involves discussions with the customer to refine the requirements. The customer then prioritizes the stories for implementation, choosing those stories that can be

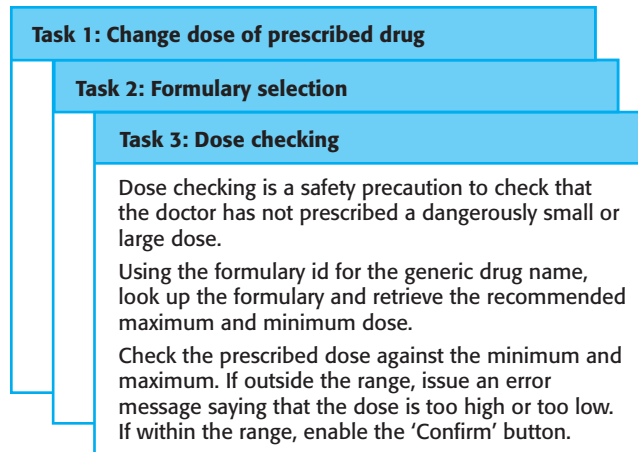


Figure 3.6 Examples of task cards for prescribing medication

used immediately to deliver useful business support. The intention is to identify useful functionality that can be implemented in about two weeks, when the next release of the system is made available to the customer.

Of course, as requirements change, the unimplemented stories change or may be discarded. If changes are required for a system that has already been delivered, new story cards are developed and again, the customer decides whether these changes should have priority over new functionality.

The idea of user stories is a powerful one—people find it much easier to relate to these stories than to a conventional requirements document or use cases. User stories can be helpful in getting users involved in suggesting requirements during an initial predevelopment requirements elicitation activity. I discuss this in more detail in Chapter 4.

The principal problem with user stories is completeness. It is difficult to judge if enough user stories have been developed to cover all of the essential requirements of a system. It is also difficult to judge if a single story gives a true picture of an activity. Experienced users are often so familiar with their work that they leave things out when describing it.

3.2.2 Refactoring

A fundamental precept of traditional software engineering is that you should design for change. That is, you should anticipate future changes to the software and design it so that these changes can be easily implemented. Extreme programming, however, has discarded this principle on the basis that designing for change is often wasted effort. It isn't worth taking time to add generality to a program to cope with change. Often the changes anticipated never materialize, or completely different change requests may actually be made.

Of course, in practice, changes will always have to be made to the code being developed. To make these changes easier, the developers of XP suggested that the code being developed should be constantly refactored. Refactoring (Fowler et al. 1999) means that the programming team look for possible improvements to the software and implements

them immediately. When team members see code that can be improved, they make these improvements even in situations where there is no immediate need for them.

A fundamental problem of incremental development is that local changes tend to degrade the software structure. Consequently, further changes to the software become harder and harder to implement. Essentially, the development proceeds by finding workarounds to problems, with the result that code is often duplicated, parts of the software are reused in inappropriate ways, and the overall structure degrades as code is added to the system. Refactoring improves the software structure and readability and so avoids the structural deterioration that naturally occurs when software is changed.

Examples of refactoring include the reorganization of a class hierarchy to remove duplicate code, the tidying up and renaming of attributes and methods, and the replacement of similar code sections, with calls to methods defined in a program library. Program development environments usually include tools for refactoring. These simplify the process of finding dependencies between code sections and making global code modifications.

In principle, when refactoring is part of the development process, the software should always be easy to understand and change as new requirements are proposed. In practice, this is not always the case. Sometimes development pressure means that refactoring is delayed because the time is devoted to the implementation of new functionality. Some new features and changes cannot readily be accommodated by code-level refactoring and require that the architecture of the system be modified.

3.2.3 Test-first development

As I discussed in the introduction to this chapter, one of the important differences between incremental development and plan-driven development is in the way that the system is tested. With incremental development, there is no system specification that can be used by an external testing team to develop system tests. As a consequence, some approaches to incremental development have a very informal testing process, in comparison with plan-driven testing.

Extreme Programming developed a new approach to program testing to address the difficulties of testing without a specification. Testing is automated and is central to the development process, and development cannot proceed until all tests have been successfully executed. The key features of testing in XP are:

1. test-first development,
2. incremental test development from scenarios,
3. user involvement in the test development and validation, and
4. the use of automated testing frameworks.

XP's test-first philosophy has now evolved into more general test-driven development techniques (Jeffries and Melnik 2007). I believe that test-driven development is one of the most important innovations in software engineering. Instead of writing code and then writing tests for that code, you write the tests before you write the code. This