SWEBOK® V3.0

Guide to the Software Engineering Body of Knowledge

Editors

Pierre Bourque
Richard E. (Dick) Fairley

Guide to the Software Engineering Body of Knowledge

Version 3.0

SWEBOK®

A Project of the IEEE Computer Society

Guide to the Software Engineering Body of Knowledge

Version 3.0

Editors

Pierre Bourque, École de technologie supérieure (ÉTS)
Richard E. (Dick) Fairley, Software and Systems Engineering Associates (S2EA)

IEEE computer society

Paperback ISBN-10: 0-7695-5166-1
Paperback ISBN-13: 978-0-7695-5166-1

Digital copies of *SWEBOK Guide* V3.0 may be downloaded free of charge for personal and academic use via www.swebok.org.

IEEE Computer Society Staff for This Publication
Angela Burgess, Executive Director
Anne Marie Kelly, Associate Executive Director, Director of Governance
Evan M. Butterfield, Director of Products and Services
John Keppler, Senior Manager, Professional Education
Kate Guillemette, Product Development Editor
Dorian McClenahan, Education Program Product Developer
Michelle Phon, Professional Education & Certification Program Coordinator
Jennie Zhu-Mai, Editorial Designer

IEEE Computer Society Products and Services. The world-renowned IEEE Computer Society publishes, promotes, and distributes a wide variety of authoritative computer science and engineering journals, magazines, conference proceedings, and professional education products. Visit the Computer Society at www.computer.org for more information.

TABLE OF CONTENTS

FOREWORD

Every profession is based on a body of knowledge, although that knowledge is not always defined in a concise manner. In cases where no formality exists, the body of knowledge is "generally recognized" by practitioners and may be codified in a variety of ways for a variety of different uses. But in many cases, a guide to a body of knowledge is formally documented, usually in a form that permits it to be used for such purposes as development and accreditation of academic and training programs, certification of specialists, or professional licensing. Generally, a professional society or similar body maintains stewardship of the formal definition of a body of knowledge.

During the past forty-five years, software engineering has evolved from a conference catchphrase into an engineering profession, characterized by 1) a professional society, 2) standards that specify generally accepted professional practices, 3) a code of ethics, 4) conference proceedings, 5) textbooks, 6) curriculum guidelines and curricula, 7) accreditation criteria and accredited degree programs, 8) certification and licensing, and 9) this Guide to the Body of Knowledge.

In this *Guide to the Software Engineering Body of Knowledge*, the IEEE Computer Society presents a revised and updated version of the body of knowledge formerly documented as SWEBOK 2004; this revised and updated version is denoted SWEBOK V3. This work is in partial fulfillment of the Society's responsibility to promote the advancement of both theory and practice for the profession of software engineering.

It should be noted that this *Guide* does not present the entire the body of knowledge for software engineering but rather serves as a guide to the body of knowledge that has been developed over more than four decades. The software engineering body of knowledge is constantly evolving. Nevertheless, this *Guide* constitutes a valuable characterization of the software engineering profession.

In 1958, John Tukey, the world-renowned statistician, coined the term *software*. The term software engineering was used in the title of a NATO conference held in Germany in 1968. The IEEE Computer Society first published its *Transactions on Software Engineering* in 1972, and a committee for developing software engineering standards was established within the IEEE Computer Society in 1976.

In 1990, planning was begun for an international standard to provide an overall view of software engineering. The standard was completed in 1995 with designation ISO/IEC 12207 and given the title of *Standard for Software Life Cycle Processes*. The IEEE version of 12207 was published in 1996 and provided a major foundation for the body of knowledge captured in SWEBOK 2004. The current version of 12207 is designated as ISO/IEC 12207:2008 and IEEE 12207-2008; it provides the basis for this SWEBOK V3.

This *Guide to the Software Engineering Body of Knowledge* is presented to you, the reader, as a mechanism for acquiring the knowledge you need in your lifelong career development as a software engineering professional.

Dick Fairley, Chair
Software and Systems Engineering Committee
IEEE Computer Society

Don Shafer, Vice President
Professional Activities Board
IEEE Computer Society

FOREWORD TO THE 2004 EDITION

In this *Guide*, the IEEE Computer Society establishes for the first time a baseline for the body of knowledge for the field of software engineering, and the work partially fulfills the Society's responsibility to promote the advancement of both theory and practice in this field. In so doing, the Society has been guided by the experience of disciplines with longer histories but was not bound either by their problems or their solutions.

It should be noted that the *Guide* does not purport to define the body of knowledge but rather to serve as a compendium and guide to the body of knowledge that has been developing and evolving over the past four decades. Furthermore, this body of knowledge is not static. The *Guide* must, necessarily, develop and evolve as software engineering matures. It nevertheless constitutes a valuable element of the software engineering infrastructure.

In 1958, John Tukey, the world-renowned statistician, coined the term *software*. The term *software engineering* was used in the title of a NATO conference held in Germany in 1968. The IEEE Computer Society first published its *Transactions on Software Engineering* in 1972. The committee established within the IEEE Computer Society for developing software engineering standards was founded in 1976.

The first holistic view of software engineering to emerge from the IEEE Computer Society resulted from an effort led by Fletcher Buckley to develop IEEE standard 730 for software quality assurance, which was completed in 1979. The purpose of IEEE Std. 730 was to provide uniform, minimum acceptable requirements for preparation and content of software quality assurance plans. This standard was influential in completing the developing standards in the following topics: configuration management, software testing, software requirements, software design, and software verification and validation.

During the period 1981–1985, the IEEE Computer Society held a series of workshops concerning the application of software engineering standards. These workshops involved practitioners sharing their experiences with existing standards. The workshops also held sessions on planning for future standards, including one involving measures and metrics for software engineering products and processes. The planning also resulted in IEEE Std. 1002, *Taxonomy of Software Engineering Standards* (1986), which provided a new, holistic view of software engineering. The standard describes the form and content of a software engineering standards taxonomy. It explains the various types of software engineering standards, their functional and external relationships, and the role of various functions participating in the software life cycle.

In 1990, planning for an international standard with an overall view was begun. The planning focused on reconciling the software process views from IEEE Std. 1074 and the revised US DoD standard 2167A. The revision was eventually published as DoD Std. 498. The international standard was completed in 1995 with designation, ISO/IEC 12207, and given the title of *Standard for Software Life Cycle Processes*. Std. ISO/IEC 12207 provided a major point of departure for the body of knowledge captured in this book.

It was the IEEE Computer Society Board of Governors' approval of the motion put forward in May 1993 by Fletcher Buckley which resulted in the writing of this book. The Association for Computing Machinery (ACM) Council approved a related motion in August 1993. The two motions led to a joint committee under the leadership of Mario Barbacci and Stuart Zweben who served as cochairs. The mission statement of the joint committee was "To establish the appropriate sets(s) of criteria and norms for professional practice of software engineering upon which industrial decisions, professional certification, and educational curricula can be based." The steering committee organized task forces in the following areas:

1. Define Required Body of Knowledge and Recommended Practices.

2. Define Ethics and Professional Standards.
3. Define Educational Curricula for undergraduate, graduate, and continuing education.

This book supplies the first component: required body of knowledge and recommend practices.

The code of ethics and professional practice for software engineering was completed in 1998 and approved by both the ACM Council and the IEEE Computer Society Board of Governors. It has been adopted by numerous corporations and other organizations and is included in several recent textbooks.

The educational curriculum for undergraduates is being completed by a joint effort of the IEEE Computer Society and the ACM and is expected to be completed in 2004.

Every profession is based on a body of knowledge and recommended practices, although they are not always defined in a precise manner. In many cases, these are formally documented, usually in a form that permits them to be used for such purposes as accreditation of academic programs, development of education and training programs, certification of specialists, or professional licensing. Generally, a professional society or related body maintains custody of such a formal definition. In cases where no such formality exists, the body of knowledge and recommended practices are "generally recognized" by practitioners and may be codified in a variety of ways for different uses.

It is hoped that readers will find this book useful in guiding them toward the knowledge and resources they need in their lifelong career development as software engineering professionals.

The book is dedicated to Fletcher Buckley in recognition of his commitment to promoting software engineering as a professional discipline and his excellence as a software engineering practitioner in radar applications.

Leonard L. Tripp, IEEE Fellow 2003
Chair, Professional Practices Committee, IEEE Computer Society (2001–2003)

Chair, Joint IEEE Computer Society and ACM Steering Committee for the Establishment of Software Engineering as a Profession (1998–1999)

Chair, Software Engineering Standards Committee, IEEE Computer Society (1992–1998)

EDITORS

Pierre Bourque, Department of Software and IT Engineering, École de technologie supérieure (ÉTS), Canada, pierre.bourque@etsmtl.ca
Richard E. (Dick) Fairley, Software and Systems Engineering Associates (S2EA), USA, dickfairley@gmail.com

COEDITORS

Alain Abran, Department of Software and IT Engineering, École de technologie supérieure (ÉTS), Canada, alain.abran@etsmtl.ca
Juan Garbajosa, Universidad Politecnica de Madrid (Technical University of Madrid, UPM), Spain, juan.garbajosa@upm.es
Gargi Keeni, Tata Consultancy Services, India, gargi@ieee.org
Beijun Shen, School of Software, Shanghai Jiao Tong University, China, bjshen@sjtu.edu.cn

CONTRIBUTING EDITORS

The following persons contributed to editing the *SWEBOK Guide* V3:
Don Shafer
Linda Shafer
Mary Jane Willshire
Kate Guillemette

CHANGE CONTROL BOARD

The following persons served on the *SWEBOK Guide* V3 Change Control Board:
Pierre Bourque
Richard E. (Dick) Fairley, Chair
Dennis Frailey
Michael Gayle
Thomas Hilburn
Paul Joannou
James W. Moore
Don Shafer
Steve Tockey

KNOWLEDGE AREA EDITORS

Software Requirements

Gerald Kotonya, School of Computing and Communications, Lancaster University, UK, gerald@comp.lancs.ac.uk
Peter Sawyer, School of Computing and Communications, Lancaster University, UK, sawyer@comp.lancs.ac.uk

Software Design

Yanchun Sun, School of Electronics Engineering and Computer Science, Peking University, China, sunyc@pku.edu.cn

Software Construction

Xin Peng, Software School, Fudan University, China, pengxin@fudan.edu.cn

Software Testing

Antonia Bertolino, ISTI-CNR, Italy, antonia.bertolino@isti.cnr.it
Eda Marchetti, ISTI-CNR, Italy, eda.marchetti@isti.cnr.it

Software Maintenance

Alain April, École de technologie supérieure (ÉTS), Canada, alain.april@etsmtl.ca
Mira Kajko-Mattsson, School of Information and Communication Technology, KTH Royal Institute of Technology, mekm2@kth.se

Software Configuration Management

Roger Champagne, École de technologie supérieure (ÉTS), Canada, roger.champagne@etsmtl.ca
Alain April, École de technologie supérieure (ÉTS), Canada, alain.april@etsmtl.ca

Software Engineering Management

James McDonald, Department of Computer Science and Software Engineering, Monmouth University, USA, jamesmc@monmouth.edu

Software Engineering Process

Annette Reilly, Lockheed Martin Information Systems & Global Solutions, USA, annette.reilly@computer.org
Richard E. Fairley, Software and Systems Engineering Associates (S2EA), USA, dickfairley@gmail.com

Software Engineering Models and Methods

Michael F. Siok, Lockheed Martin Aeronautics Company, USA, mike.f.siok@lmco.com

Software Quality

J. David Blaine, USA, jdavidblaine@gmail.com
Durba Biswas, Tata Consultancy Services, India, durba.biswas@tcs.com

Software Engineering Professional Practice
Aura Sheffield, USA, arsheff@acm.org
Hengming Zou, Shanghai Jiao Tong University, China, zou@sjtu.edu.cn

Software Engineering Economics
Christof Ebert, Vector Consulting Services, Germany, christof.ebert@vector.com

Computing Foundations
Hengming Zou, Shanghai Jiao Tong University, China, zou@sjtu.edu.cn

Mathematical Foundations
Nabendu Chaki, University of Calcutta, India, nabendu@ieee.org

Engineering Foundations
Amitava Bandyopadhayay, Indian Statistical Institute, India, bamitava@isical.ac.in
Mary Jane Willshire, Software and Systems Engineering Associates (S2EA), USA, mj.fairley@gmail.com

Appendix B: IEEE and ISO/IEC Standards Supporting SWEBOK
James W. Moore, USA, James.W.Moore@ieee.org

KNOWLEDGE AREA EDITORS OF PREVIOUS SWEBOK VERSIONS

The following persons served as Associate Editors for either the Trial version published in 2001 or for the 2004 version.

Software Requirements

Peter Sawyer, Computing Department, Lancaster University, UK
Gerald Kotonya, Computing Department, Lancaster University, UK

Software Design

Guy Tremblay, Département d'informatique, UQAM, Canada

Software Construction

Steve McConnell, Construx Software, USA
Terry Bollinger, the MITRE Corporation, USA
Philippe Gabrini, Département d'informatique, UQAM, Canada
Louis Martin, Département d'informatique, UQAM, Canada

Software Testing

Antonia Bertolino, ISTI-CNR, Italy
Eda Marchetti, ISTI-CNR, Italy

Software Maintenance

Thomas M. Pigoski, Techsoft Inc., USA
Alain April, École de technologie supérieure, Canada

Software Configuration Management

John A. Scott, Lawrence Livermore National Laboratory, USA
David Nisse, USA

Software Engineering Management

Dennis Frailey, Raytheon Company, USA
Stephen G. MacDonell, Auckland University of Technology, New Zealand
Andrew R. Gray, University of Otago, New Zealand

Software Engineering Process

Khaled El Emam, served while at the Canadian National Research Council, Canada

Software Engineering Tools and Methods

David Carrington, School of Information Technology and Electrical Engineering,
The University of Queensland, Australia

Software Quality

Alain April, École de technologie supérieure, Canada
Dolores Wallace, retired from the National Institute of Standards and Technology, USA
Larry Reeker, NIST, USA

References Editor

Marc Bouisset, Département d'informatique, UQAM

REVIEW TEAM

The people listed below participated in the public review process of *SWEBOK Guide* V3. Membership of the IEEE Computer Society was not a requirement to participate in this review process, and membership information was not requested from reviewers. Over 1500 individual comments were collected and duly adjudicated.

Carlos C. Amaro, USA
Mark Ardis, USA
Mora-Soto Arturo, Spain
Ohad Barzilay, Israel
Gianni Basaglia, Italy
Denis J. Bergquist, USA
Alexander Bogush, UK
Christopher Bohn, USA
Steve Bollweg, USA
Reto Bonderer, Switzerland
Alexei Botchkarev, Canada
Pieter Botman, Canada
Robert Bragner, USA
Kevin Brune, USA
Ogihara Bryan, USA
Luigi Buglione, Italy
Rick Cagle, USA
Barbara Canody, USA
Rogerio A. Carvalho, Brazil
Daniel Cerys, USA
Philippe Cohard, France
Ricardo Colomo-Palacios, Spain
Mauricio Coria, Argentina
Marek Cruz, UK
Stephen Danckert, USA
Bipul K. Das, Canada
James D. Davidson, USA
Jon Dehn, USA
Lincoln P. Djang, USA
Andreas Doblander, Austria
Yi-Ben Doo, USA
Scott J. Dougherty, UK
Regina DuBord, USA
Fedor Dzerzhinskiy, Russia
Ann M. Eblen, Australia
David M. Endres, USA
Marilyn Escue, USA
Varuna Eswer, India
Istvan Fay, Hungary
Jose L. Fernandez-Sanchez, Spain
Dennis J. Frailey, USA
Tihana Galinac Grbac, Croatia
Colin Garlick, New Zealand
Garth J.G. Glynn, UK
Jill Gostin, USA
Christiane Gresse von Wangenheim, Brazil
Thomas Gust, USA
H.N. Mok, Singapore
Jon D. Hagar, USA
Anees Ahmed Haidary, India
Duncan Hall, New Zealand
James Hart, USA
Jens H.J. Heidrich, Germany
Rich Hilliard, USA
Bob Hillier, Canada
Norman M. Hines, USA
Dave Hirst, USA
Theresa L. Hunt, USA
Kenneth Ingham, USA
Masahiko Ishikawa, Japan
Michael A. Jablonski, USA
G. Jagadeesh, India
Sebastian Justicia, Spain
Umut Kahramankaptan, Belgium
Pankaj Kamthan, Canada
Perry Kapadia, USA
Tarig A. Khalid, Sudan
Michael K.A. Klaes, Germany
Maged Koshty, Egypt
Claude C. Laporte, Canada
Dong Li, China
Ben Linders, Netherlands
Claire Lohr, USA
Vladimir Mandic, Serbia
Matt Mansell, New Zealand
John Marien, USA

Stephen P. Masticola, USA
Nancy Mead, USA
Fuensanta Medina-Dominguez, Spain
Silvia Judith Meles, Argentina
Oscar A. Mondragon, Mexico
David W. Mutschler, USA
Maria Nelson, Brazil
John Noblin, USA
Bryan G. Ogihara, USA
Takehisa Okazaki, Japan
Hanna Oktaba, Mexico
Chin Hwee Ong, Hong Kong
Venkateswar Oruganti, India
Birgit Penzenstadler, Germany
Larry Peters, USA
S.K. Pillai, India
Vaclav Rajlich, USA
Kiron Rao, India
Luis Reyes, USA
Hassan Reza, USA
Steve Roach, USA
Teresa L. Roberts, USA
Dennis Robi, USA
Warren E. Robinson, USA
Jorge L. Rodriguez, USA
Alberto C. Sampaio, Portugal
Ed Samuels, USA
Maria-Isabel Sanchez-Segura, Spain
Vineet Sawant, USA
R. Schaaf, USA
James C. Schatzman, USA
Oscar A. Schivo, Argentina
Florian Schneider, Germany
Thom Schoeffling, USA
Reinhard Schrage, Germany
Neetu Sethia, India
Cindy C. Shelton, USA
Alan Shepherd, Germany
Katsutoshi Shintani, Japan
Erik Shreve, USA
Jaguaraci Silva, Brazil
M. Somasundaram, India
Peraphon Sophatsathit, Thailand
John Standen, UK
Joyce Statz, USA
Perdita P. Stevens, UK
David Struble, USA
Ohno Susumu, Japan
Urcun Tanik, USA
Talin Tasciyan, USA
J. Barrie Thompson, UK
Steve Tockey, USA
Miguel Eduardo Torres Moreno, Colombia
Dawid Trawczynski, USA
Adam Trendowicz, Germany
Norio Ueno, Japan
Cenk Uyan, Turkey
Chandra Sekar Veerappan, Singapore
Oruganti Venkateswar, India
Jochen Vogt, Germany
Hironori Washizaki, Japan
Ulf Westermann, Germany
Don Wilson, USA
Aharon Yadin, Israel
Hong Zhou, UK

ACKNOWLEDGEMENTS

Funding for the development of *SWEBOK Guide* V3 has been provided by the IEEE Computer Society. The editors and coeditors appreciate the important work performed by the KA editors and the contributing editors as well as by the the members of the Change Control Board. The editorial team must also acknowledge the indispensable contribution of reviewers.

The editorial team also wishes to thank the following people who contributed to the project in various ways: Pieter Botman, Evan Butterfield, Carine Chauny, Pierce Gibbs, Diane Girard, John Keppler, Dorian McClenahan, Kenza Meridji, Samuel Redwine, Annette Reilly, and Pam Thompson.

Finally, there are surely other people who have contributed to this *Guide*, either directly or indirectly, whose names we have inadvertently omitted. To those people, we offer our tacit appreciation and apologize for having omitted explicit recognition.

IEEE COMPUTER SOCIETY PRESIDENTS

Dejan Milojicic, 2014 President
David Alan Grier, 2013 President
Thomas Conte, 2015 President

PROFESSIONAL ACTIVITIES BOARD, 2013 MEMBERSHIP

Donald F. Shafer, Chair
Pieter Botman, CSDP
Pierre Bourque
Richard Fairley, CSDP
Dennis Frailey
S. Michael Gayle
Phillip Laplante, CSDP
Jim Moore, CSDP
Linda Shafer, CSDP
Steve Tockey, CSDP
Charlene “Chuck” Walrad

MOTIONS REGARDING THE APPROVAL OF SWEBOK GUIDE V3.0

The *SWEBOK Guide* V3.0 was submitted to ballot by verified IEEE Computer Society members in November 2013 with the following question: "Do you approve this manuscript of the *SWEBOK Guide* V3.0 to move forward to formatting and publication?"

The results of this ballot were 259 Yes votes and 5 No votes.

The following motion was unanimously adopted by the Professional Activities Board of the IEEE Computer Society in December 2013:

> *The Professional Activities Board of the IEEE Computer Society finds that the* Guide to the Software Engineering Body of Knowledge *Version 3.0 has been successfully completed; and endorses the* Guide to the Software Engineering Body of Knowledge *Version 3.0 and commends it to the IEEE Computer Society Board of Governors for their approval.*

The following motion was adopted by the IEEE Computer Society Board of Governors in December 2013:

> *MOVED, that the Board of Governors of the IEEE Computer Society approves Version 3.0 of the* Guide to the Software Engineering Body of Knowledge *and authorizes the Chair of the Professional Activities Board to proceed with printing.*

MOTIONS REGARDING THE APPROVAL OF SWEBOK GUIDE 2004 VERSION

The following motion was unanimously adopted by the Industrial Advisory Board of the *SWEBOK Guide* project in February 2004:

> *The Industrial Advisory Board finds that the Software Engineering Body of Knowledge project initiated in 1998 has been successfully completed; and endorses the 2004 Version of the* Guide to the SWEBOK *and commends it to the IEEE Computer Society Board of Governors for their approval.*

The following motion was adopted by the IEEE Computer Society Board of Governors in February 2004:

> *MOVED, that the Board of Governors of the IEEE Computer Society approves the 2004 Edition of the* Guide to the Software Engineering Body of Knowledge *and authorizes the Chair of the Professional Practices Committee to proceed with printing.*

Please also note that the 2004 edition of the *Guide to the Software Engineering Body of Knowledge* was submitted by the IEEE Computer Society to ISO/IEC without any change and was recognized as Technical Report ISO/IEC TR 19759:2005.

INTRODUCTION TO THE GUIDE

KA	Knowledge Area
SWEBOK	Software Engineering Body of Knowledge

Publication of the 2004 version of this *Guide to the Software Engineering Body of Knowledge* (SWEBOK 2004) was a major milestone in establishing software engineering as a recognized engineering discipline. The goal in developing this update to SWEBOK is to improve the currency, readability, consistency, and usability of the *Guide*.

All knowledge areas (KAs) have been updated to reflect changes in software engineering since publication of SWEBOK 2004. Four new foundation KAs and a Software Engineering Professional Practices KA have been added. The Software Engineering Tools and Methods KA has been revised as Software Engineering Models and Methods. Software engineering tools is now a topic in each of the KAs. Three appendices provide the specifications for the KA description, an annotated set of relevant standards for each KA, and a listing of the references cited in the *Guide*.

This *Guide*, written under the auspices of the Professional Activities Board of the IEEE Computer Society, represents a next step in the evolution of the software engineering profession.

WHAT IS SOFTWARE ENGINEERING?

ISO/IEC/IEEE Systems and Software Engineering Vocabulary (SEVOCAB) defines software engineering as "the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software)."[1]

WHAT ARE THE OBJECTIVES OF THE SWEBOK GUIDE?

The *Guide* should not be confused with the Body of Knowledge itself, which exists in the published literature. The purpose of the *Guide* is to describe the portion of the Body of Knowledge that is generally accepted, to organize that portion, and to provide topical access to it.

The *Guide to the Software Engineering Body of Knowledge* (*SWEBOK Guide*) was established with the following five objectives:

1. To promote a consistent view of software engineering worldwide
2. To specify the scope of, and clarify the place of software engineering with respect to other disciplines such as computer science, project management, computer engineering, and mathematics
3. To characterize the contents of the software engineering discipline
4. To provide a topical access to the Software Engineering Body of Knowledge
5. To provide a foundation for curriculum development and for individual certification and licensing material

The first of these objectives, a consistent worldwide view of software engineering, was supported by a development process which engaged approximately 150 reviewers from 33 countries. More information regarding the development process can be found on the website (www.swebok.org). Professional and learned societies and public agencies involved in software engineering were contacted, made aware of this project to update SWEBOK, and invited to participate in the review process. KA editors were recruited from North America, the Pacific Rim, and Europe. Presentations on the project were made at various international venues.

The second of the objectives, the desire to specify the scope of software engineering, motivates the fundamental organization of the Guide. The material that is recognized as being within this discipline is organized into the fifteen KAs listed in Table I.1. Each of these KAs is treated in a chapter in this *Guide*.

1 See www.computer.org/sevocab.

Table I.1. The 15 SWEBOK KAs
Software Requirements
Software Design
Software Construction
Software Testing
Software Maintenance
Software Configuration Management
Software Engineering Management
Software Engineering Process
Software Engineering Models and Methods
Software Quality
Software Engineering Professional Practice
Software Engineering Economics
Computing Foundations
Mathematical Foundations
Engineering Foundations

In specifying scope, it is also important to identify the disciplines that intersect with software engineering. To this end, SWEBOK V3 also recognizes seven related disciplines, listed in Table I.2. Software engineers should, of course, have knowledge of material from these disciplines (and the KA descriptions in this *Guide* may make reference to them). It is not, however, an objective of the *SWEBOK Guide* to characterize the knowledge of the related disciplines.

Table I.2. Related Disciplines
Computer Engineering
Computer Science
General Management
Mathematics
Project Management
Quality Management
Systems Engineering

The relevant elements of computer science and mathematics are presented in the Computing Foundations and Mathematical Foundations KAs of the *Guide* (Chapters 13 and 14).

HIERARCHICAL ORGANIZATION

The organization of the KA chapters supports the third of the project's objectives—a characterization of the contents of software engineering. The detailed specifications provided by the project's editorial team to the associate editors regarding the contents of the KA descriptions can be found in Appendix A.

The *Guide* uses a hierarchical organization to decompose each KA into a set of topics with recognizable labels. A two (sometime three) level breakdown provides a reasonable way to find topics of interest. The *Guide* treats the selected topics in a manner compatible with major schools of thought and with breakdowns generally found in industry and in software engineering literature and standards. The breakdowns of topics do not presume particular application domains, business uses, management philosophies, development methods, and so forth. The extent of each topic's description is only that needed to understand the generally accepted nature of the topics and for the reader to successfully find reference material; the Body of Knowledge is found in the reference materials themselves, not in the *Guide*.

REFERENCE MATERIAL AND MATRIX

To provide topical access to the knowledge—the fourth of the project's objectives—the *Guide* identifies authoritative reference material for each KA. Appendix C provides a Consolidated Reference List for the *Guide*. Each KA includes relevant references from the Consolidated Reference List and also includes a matrix relating the reference material to the included topics.

It should be noted that the *Guide* does not attempt to be comprehensive in its citations. Much material that is both suitable and excellent is not referenced. Material included in the Consolidated Reference List provides coverage of the topics described.

DEPTH OF TREATMENT

To achieve the SWEBOK fifth objective—providing a foundation for curriculum development,

certification, and licensing, the criterion of *generally accepted* knowledge has been applied, to be distinguished from advanced and research knowledge (on the grounds of maturity) and from specialized knowledge (on the grounds of generality of application).

The equivalent term *generally recognized* comes from the Project Management Institute: "Generally recognized means the knowledge and practices described are applicable to most projects most of the time, and there is consensus about their value and usefulness."[2]

However, the terms "generally accepted" or "generally recognized" do not imply that the designated knowledge should be uniformly applied to all software engineering endeavors—each project's needs determine that—but it does imply that competent, capable software engineers should be equipped with this knowledge for potential application. More precisely, generally accepted knowledge should be included in the study material for the software engineering licensing examination that graduates would take after gaining four years of work experience. Although this criterion is specific to the US style of education and does not necessarily apply to other countries, we deem it useful.

STRUCTURE OF THE KA DESCRIPTIONS

The KA descriptions are structured as follows.

In the introduction, a brief definition of the KA and an overview of its scope and of its relationship with other KAs are presented.

The breakdown of topics in each KA constitutes the core the KA description, describing the decomposition of the KA into subareas, topics, and sub-topics. For each topic or subtopic, a short description is given, along with one or more references.

The reference material was chosen because it is considered to constitute the best presentation of the knowledge relative to the topic. A matrix links the topics to the reference material.

The last part of each KA description is the list of recommended references and (optionally) further readings. Relevant standards for each KA are presented in Appendix B of the *Guide*.

APPENDIX A. KA DESCRIPTION SPECIFICATIONS

Appendix A describes the specifications provided by the editorial team to the associate editors for the content, recommended references, format, and style of the KA descriptions.

APPENDIX B. ALLOCATION OF STANDARDS TO KAS

Appendix B is an annotated list of the relevant standards, mostly from the IEEE and the ISO, for each of the KAs of the *SWEBOK Guide*.

APPENDIX C. CONSOLIDATED REFERENCE LIST

Appendix C contains the consolidated list of recommended references cited in the KAs (these references are marked with an asterisk (*) in the text).

2 *A Guide to the Project Management Body of Knowledge*, 5th ed., Project Management Institute, 2013; www.pmi.org.

CHAPTER 1

SOFTWARE REQUIREMENTS

ACRONYMS

CIA	Confidentiality, Integrity, and Availability
DAG	Directed Acyclic Graph
FSM	Functional Size Measurement
INCOSE	International Council on Systems Engineering
UML	Unified Modeling Language
SysML	Systems Modeling Language

INTRODUCTION

The Software Requirements knowledge area (KA) is concerned with the elicitation, analysis, specification, and validation of software requirements as well as the management of requirements during the whole life cycle of the software product. It is widely acknowledged amongst researchers and industry practitioners that software projects are critically vulnerable when the requirements-related activities are poorly performed.

Software requirements express the needs and constraints placed on a software product that contribute to the solution of some real-world problem.

The term "requirements engineering" is widely used in the field to denote the systematic handling of requirements. For reasons of consistency, the term "engineering" will not be used in this KA other than for software engineering per se.

For the same reason, "requirements engineer," a term which appears in some of the literature, will not be used either. Instead, the term "software engineer" or, in some specific cases, "requirements specialist" will be used, the latter where the role in question is usually performed by an individual other than a software engineer. This does not imply, however, that a software engineer could not perform the function.

A risk inherent in the proposed breakdown is that a waterfall-like process may be inferred. To guard against this, topic 2, Requirements Process, is designed to provide a high-level overview of the requirements process by setting out the resources and constraints under which the process operates and which act to configure it.

An alternate decomposition could use a product-based structure (system requirements, software requirements, prototypes, use cases, and so on). The process-based breakdown reflects the fact that the requirements process, if it is to be successful, must be considered as a process involving complex, tightly coupled activities (both sequential and concurrent), rather than as a discrete, one-off activity performed at the outset of a software development project.

The Software Requirements KA is related closely to the Software Design, Software Testing, Software Maintenance, Software Configuration Management, Software Engineering Management, Software Engineering Process, Software Engineering Models and Methods, and Software Quality KAs.

BREAKDOWN OF TOPICS FOR SOFTWARE REQUIREMENTS

The breakdown of topics for the Software Requirements KA is shown in Figure 1.1.

1. Software Requirements Fundamentals

[1*, c4, c4s1, c10s1, c10s4] [2*, c1, c6, c12]

1.1. Definition of a Software Requirement

At its most basic, a software requirement is a property that must be exhibited by something in

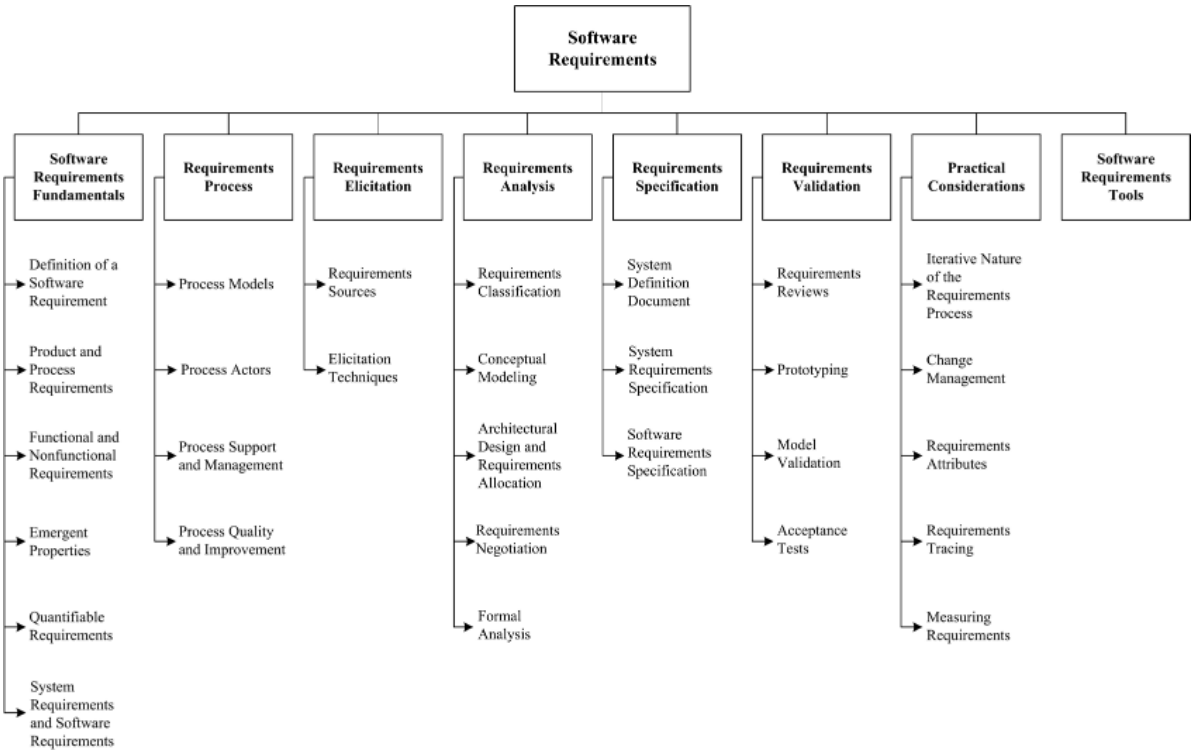

Figure 1.1. Breakdown of Topics for the Software Requirements KA

order to solve some problem in the real world. It may aim to automate part of a task for someone to support the business processes of an organization, to correct shortcomings of existing software, or to control a device—to name just a few of the many problems for which software solutions are possible. The ways in which users, business processes, and devices function are typically complex. By extension, therefore, the requirements on particular software are typically a complex combination from various people at different levels of an organization, and who are in one way or another involved or connected with this feature from the environment in which the software will operate.

An essential property of all software requirements is that they be verifiable as an individual feature as a functional requirement or at the system level as a nonfunctional requirement. It may be difficult or costly to verify certain software requirements. For example, verification of the throughput requirement on a call center may necessitate the development of simulation software. Software requirements, software testing, and quality personnel must ensure that the requirements can be verified within available resource constraints.

Requirements have other attributes in addition to behavioral properties. Common examples include a priority rating to enable tradeoffs in the face of finite resources and a status value to enable project progress to be monitored. Typically, software requirements are uniquely identified so that they can be subjected to software configuration management over the entire life cycle of the feature and of the software.

1.2. Product and Process Requirements

A product requirement is a need or constraint on the software to be developed (for example, "The software shall verify that a student meets all prerequisites before he or she registers for a course").

A process requirement is essentially a constraint on the development of the software (for example, "The software shall be developed using a RUP process").

Some software requirements generate implicit process requirements. The choice of verification

technique is one example. Another might be the use of particularly rigorous analysis techniques (such as formal specification methods) to reduce faults that can lead to inadequate reliability. Process requirements may also be imposed directly by the development organization, their customer, or a third party such as a safety regulator.

1.3. Functional and Nonfunctional Requirements

Functional requirements describe the functions that the software is to execute; for example, formatting some text or modulating a signal. They are sometimes known as capabilities or features. A functional requirement can also be described as one for which a finite set of test steps can be written to validate its behavior.

Nonfunctional requirements are the ones that act to constrain the solution. Nonfunctional requirements are sometimes known as constraints or quality requirements. They can be further classified according to whether they are performance requirements, maintainability requirements, safety requirements, reliability requirements, security requirements, interoperability requirements or one of many other types of software requirements (see Models and Quality Characteristics in the Software Quality KA).

1.4. Emergent Properties

Some requirements represent emergent properties of software—that is, requirements that cannot be addressed by a single component but that depend on how all the software components interoperate. The throughput requirement for a call center would, for example, depend on how the telephone system, information system, and the operators all interacted under actual operating conditions. Emergent properties are crucially dependent on the system architecture.

1.5. Quantifiable Requirements

Software requirements should be stated as clearly and as unambiguously as possible, and, where appropriate, quantitatively. It is important to avoid vague and unverifiable requirements that depend for their interpretation on subjective judgment ("the software shall be reliable"; "the software shall be user-friendly"). This is particularly important for nonfunctional requirements. Two examples of quantified requirements are the following: a call center's software must increase the center's throughput by 20%; and a system shall have a probability of generating a fatal error during any hour of operation of less than $1 * 10^{-8}$. The throughput requirement is at a very high level and will need to be used to derive a number of detailed requirements. The reliability requirement will tightly constrain the system architecture.

1.6. System Requirements and Software Requirements

In this topic, "system" means

> an interacting combination of elements to accomplish a defined objective. These include hardware, software, firmware, people, information, techniques, facilities, services, and other support elements,

as defined by the International Council on Software and Systems Engineering (INCOSE) [3].

System requirements are the requirements for the system as a whole. In a system containing software components, *software* requirements are derived from system requirements.

This KA defines "user requirements" in a restricted way, as the requirements of the system's customers or end users. System requirements, by contrast, encompass user requirements, requirements of other stakeholders (such as regulatory authorities), and requirements without an identifiable human source.

2. Requirements Process

[1*, c4s4] [2*, c1–4, c6, c22, c23]

This section introduces the software requirements process, orienting the remaining five topics and showing how the requirements process dovetails with the overall software engineering process.

2.1. Process Models

The objective of this topic is to provide an understanding that the requirements process

- is not a discrete front-end activity of the software life cycle, but rather a process initiated at the beginning of a project that continues to be refined throughout the life cycle;
- identifies software requirements as configuration items and manages them using the same software configuration management practices as other products of the software life cycle processes;
- needs to be adapted to the organization and project context.

In particular, the topic is concerned with how the activities of elicitation, analysis, specification, and validation are configured for different types of projects and constraints. The topic also includes activities that provide input into the requirements process, such as marketing and feasibility studies.

2.2. Process Actors

This topic introduces the roles of the people who participate in the requirements process. This process is fundamentally interdisciplinary, and the requirements specialist needs to mediate between the domain of the stakeholder and that of software engineering. There are often many people involved besides the requirements specialist, each of whom has a stake in the software. The stakeholders will vary across projects, but will always include users/operators and customers (who need not be the same).

Typical examples of software stakeholders include (but are not restricted to) the following:

- Users: This group comprises those who will operate the software. It is often a heterogeneous group involving people with different roles and requirements.
- Customers: This group comprises those who have commissioned the software or who represent the software's target market.
- Market analysts: A mass-market product will not have a commissioning customer, so marketing people are often needed to establish what the market needs and to act as proxy customers.
- Regulators: Many application domains, such as banking and public transport, are regulated. Software in these domains must comply with the requirements of the regulatory authorities.
- Software engineers: These individuals have a legitimate interest in profiting from developing the software by, for example, reusing components in or from other products. If, in this scenario, a customer of a particular product has specific requirements that compromise the potential for component reuse, the software engineers must carefully weigh their own stake against those of the customer. Specific requirements, particularly constraints, may have major impact on project cost or delivery because they either fit well or poorly with the skill set of the engineers. Important tradeoffs among such requirements should be identified.

It will not be possible to perfectly satisfy the requirements of every stakeholder, and it is the software engineer's job to negotiate tradeoffs that are both acceptable to the principal stakeholders and within budgetary, technical, regulatory, and other constraints. A prerequisite for this is that all the stakeholders be identified, the nature of their "stake" analyzed, and their requirements elicited.

2.3. Process Support and Management

This section introduces the project management resources required and consumed by the requirements process. It establishes the context for the first topic (Initiation and Scope Definition) of the Software Engineering Management KA. Its principal purpose is to make the link between the process activities identified in 2.1 and the issues of cost, human resources, training, and tools.

2.4. Process Quality and Improvement

This topic is concerned with the assessment of the quality and improvement of the requirements process. Its purpose is to emphasize the key role the requirements process plays in terms of the

cost and timeliness of a software product and of the customer's satisfaction with it. It will help to orient the requirements process with quality standards and process improvement models for software and systems. Process quality and improvement is closely related to both the Software Quality KA and Software Engineering Process KA, comprising

- requirements process coverage by process improvement standards and models;
- requirements process measures and benchmarking;
- improvement planning and implementation;
- security/CIA improvement/planning and implementation.

3. Requirements Elicitation

[1*, c4s5] [2*, c5, c6, c9]

Requirements elicitation is concerned with the origins of software requirements and how the software engineer can collect them. It is the first stage in building an understanding of the problem the software is required to solve. It is fundamentally a human activity and is where the stakeholders are identified and relationships established between the development team and the customer. It is variously termed "requirements capture," "requirements discovery," and "requirements acquisition."

One of the fundamental principles of a good requirements elicitation process is that of effective communication between the various stakeholders. This communication continues through the entire Software Development Life Cycle (SDLC) process with different stakeholders at different points in time. Before development begins, requirements specialists may form the conduit for this communication. They must mediate between the domain of the software users (and other stakeholders) and the technical world of the software engineer. A set of internally consistent models at different levels of abstraction facilitate communications between software users/stakeholders and software engineers.

A critical element of requirements elicitation is informing the project scope. This involves providing a description of the software being specified and its purpose and prioritizing the deliverables to ensure the customer's most important business needs are satisfied first. This minimizes the risk of requirements specialists spending time eliciting requirements that are of low importance, or those that turn out to be no longer relevant when the software is delivered. On the other hand, the description must be scalable and extensible to accept further requirements not expressed in the first formal lists and compatible with the previous ones as contemplated in recursive methods.

3.1. Requirements Sources

Requirements have many sources in typical software, and it is essential that all potential sources be identified and evaluated. This topic is designed to promote awareness of the various sources of software requirements and of the frameworks for managing them. The main points covered are as follows:

- Goals. The term "goal" (sometimes called "business concern" or "critical success factor") refers to the overall, high-level objectives of the software. Goals provide the motivation for the software but are often vaguely formulated. Software engineers need to pay particular attention to assessing the value (relative to priority) and cost of goals. A feasibility study is a relatively low-cost way of doing this.
- Domain knowledge. The software engineer needs to acquire or have available knowledge about the application domain. Domain knowledge provides the background against which all elicited requirements knowledge must be set in order to understand it. It's a good practice to emulate an ontological approach in the knowledge domain. Relations between relevant concepts within the application domain should be identified.
- Stakeholders (see section 2.2, Process Actors). Much software has proved unsatisfactory because it has stressed the requirements of one group of stakeholders at the expense of others. Hence, the delivered software is difficult to use, or subverts the cultural or political structures of the customer organization. The software engineer needs to identify, represent, and manage

the "viewpoints" of many different types of stakeholders.

- Business rules. These are statements that define or constrain some aspect of the structure or the behavior of the business itself. "A student cannot register in next semester's courses if there remain some unpaid tuition fees" would be an example of a business rule that would be a requirement source for a university's course-registration software.
- The operational environment. Requirements will be derived from the environment in which the software will be executed. These may be, for example, timing constraints in real-time software or performance constraints in a business environment. These must be sought out actively because they can greatly affect software feasibility and cost as well as restrict design choices.
- The organizational environment. Software is often required to support a business process, the selection of which may be conditioned by the structure, culture, and internal politics of the organization. The software engineer needs to be sensitive to these since, in general, new software should not force unplanned change on the business process.

3.2. Elicitation Techniques

Once the requirements sources have been identified, the software engineer can start eliciting requirements information from them. Note that requirements are seldom elicited ready-made. Rather, the software engineer elicits information from which he or she formulates requirements. This topic concentrates on techniques for getting human stakeholders to articulate requirements-relevant information. It is a very difficult task and the software engineer needs to be sensitized to the fact that (for example) users may have difficulty describing their tasks, may leave important information unstated, or may be unwilling or unable to cooperate. It is particularly important to understand that elicitation is not a passive activity and that, even if cooperative and articulate stakeholders are available, the software engineer has to work hard to elicit the right information. Many business or technical requirements are tacit or in feedback that has yet to be obtained from end users. The importance of planning, verification, and validation in requirements elicitation cannot be overstated. A number of techniques exist for requirements elicitation; the principal ones are these:

- Interviews. Interviewing stakeholders is a "traditional" means of eliciting requirements. It is important to understand the advantages and limitations of interviews and how they should be conducted.
- Scenarios. Scenarios provide a valuable means for providing context to the elicitation of user requirements. They allow the software engineer to provide a framework for questions about user tasks by permitting "what if" and "how is this done" questions to be asked. The most common type of scenario is the use case description. There is a link here to topic 4.2 (Conceptual Modeling) because scenario notations such as use case diagrams are common in modeling software.
- Prototypes. This technique is a valuable tool for clarifying ambiguous requirements. They can act in a similar way to scenarios by providing users with a context within which they can better understand what information they need to provide. There is a wide range of prototyping techniques—from paper mock-ups of screen designs to beta-test versions of software products—and a strong overlap of their separate uses for requirements elicitation and for requirements validation (see section 6.2, Prototyping). Low fidelity prototypes are often preferred to avoid stakeholder "anchoring" on minor, incidental characteristics of a higher quality prototype that can limit design flexibility in unintended ways.
- Facilitated meetings. The purpose of these meetings is to try to achieve a summative effect, whereby a group of people can bring more insight into their software requirements than by working individually. They can brainstorm and refine ideas that may be difficult to bring to the surface using interviews. Another advantage is that conflicting requirements surface early on in a way that lets the stakeholders recognize where these occur. When it works well, this technique

may result in a richer and more consistent set of requirements than might otherwise be achievable. However, meetings need to be handled carefully (hence the need for a facilitator) to prevent a situation in which the critical abilities of the team are eroded by group loyalty, or in which requirements reflecting the concerns of a few outspoken (and perhaps senior) people that are favored to the detriment of others.

- Observation. The importance of software context within the organizational environment has led to the adaptation of observational techniques such as ethnography for requirements elicitation. Software engineers learn about user tasks by immersing themselves in the environment and observing how users perform their tasks by interacting with each other and with software tools and other resources. These techniques are relatively expensive but also instructive because they illustrate that many user tasks and business processes are too subtle and complex for their actors to describe easily.
- User stories. This technique is commonly used in adaptive methods (see Agile Methods in the Software Engineering Models and Methods KA) and refers to short, high-level descriptions of required functionality expressed in customer terms. A typical user story has the form: *"As a <role>, I want <goal/desire> so that <benefit>."* A user story is intended to contain just enough information so that the developers can produce a reasonable estimate of the effort to implement it. The aim is to avoid some of the waste that often happens in projects where detailed requirements are gathered early but become invalid before the work begins. Before a user story is implemented, an appropriate acceptance procedure must be written by the customer to determine whether the goals of the user story have been fulfilled.
- Other techniques. A range of other techniques for supporting the elicitation of requirements information exist and range from analyzing competitors' products to applying data mining techniques to using sources of domain knowledge or customer request databases.

4. Requirements Analysis

[1*, c4s1, c4s5, c10s4, c12s5]
[2*, c7, c11, c12, c17]

This topic is concerned with the process of analyzing requirements to

- detect and resolve conflicts between requirements;
- discover the bounds of the software and how it must interact with its organizational and operational environment;
- elaborate system requirements to derive software requirements.

The traditional view of requirements analysis has been that it be reduced to conceptual modeling using one of a number of analysis methods, such as the structured analysis method. While conceptual modeling is important, we include the classification of requirements to help inform tradeoffs between requirements (requirements classification) and the process of establishing these tradeoffs (requirements negotiation).

Care must be taken to describe requirements precisely enough to enable the requirements to be validated, their implementation to be verified, and their costs to be estimated.

4.1. Requirements Classification

Requirements can be classified on a number of dimensions. Examples include the following:

- Whether the requirement is functional or nonfunctional (see section 1.3, Functional and Nonfunctional Requirements).
- Whether the requirement is derived from one or more high-level requirements or an emergent property (see section 1.4, Emergent Properties), or is being imposed directly on the software by a stakeholder or some other source.
- Whether the requirement is on the product or the process (see section 1.2, Product and Process Requirements). Requirements on the process can constrain the choice of contractor, the software engineering process to be adopted, or the standards to be adhered to.

- The requirement priority. The higher the priority, the more essential the requirement is for meeting the overall goals of the software. Often classified on a fixed-point scale such as mandatory, highly desirable, desirable, or optional, the priority often has to be balanced against the cost of development and implementation.
- The scope of the requirement. Scope refers to the extent to which a requirement affects the software and software components. Some requirements, particularly certain nonfunctional ones, have a global scope in that their satisfaction cannot be allocated to a discrete component. Hence, a requirement with global scope may strongly affect the software architecture and the design of many components, whereas one with a narrow scope may offer a number of design choices and have little impact on the satisfaction of other requirements.
- Volatility/stability. Some requirements will change during the life cycle of the software—and even during the development process itself. It is useful if some estimate of the likelihood that a requirement will change can be made. For example, in a banking application, requirements for functions to calculate and credit interest to customers' accounts are likely to be more stable than a requirement to support a particular kind of tax-free account. The former reflects a fundamental feature of the banking domain (that accounts can earn interest), while the latter may be rendered obsolete by a change to government legislation. Flagging potentially volatile requirements can help the software engineer establish a design that is more tolerant of change.

Other classifications may be appropriate, depending upon the organization's normal practice and the application itself.

There is a strong overlap between requirements classification and requirements attributes (see section 7.3, Requirements Attributes).

4.2. Conceptual Modeling

The development of models of a real-world problem is key to software requirements analysis. Their purpose is to aid in understanding the situation in which the problem occurs, as well as depicting a solution. Hence, conceptual models comprise models of entities from the problem domain, configured to reflect their real-world relationships and dependencies. This topic is closely related to the Software Engineering Models and Methods KA.

Several kinds of models can be developed. These include use case diagrams, data flow models, state models, goal-based models, user interactions, object models, data models, and many others. Many of these modeling notations are part of *the Unified Modeling Language (UML)*. Use case diagrams, for example, are routinely used to depict scenarios where the boundary separates the actors (users or systems in the external environment) from the internal behavior where each use case depicts a functionality of the system.

The factors that influence the choice of modeling notation include these:

- The nature of the problem. Some types of software demand that certain aspects be analyzed particularly rigorously. For example, state and parametric models, which are part of SysML [4], are likely to be more important for real-time software than for information systems, while it would usually be the opposite for object and activity models.
- The expertise of the software engineer. It is often more productive to adopt a modeling notation or method with which the software engineer has experience.
- The process requirements of the customer (see section 1.2, Product and Process Requirements). Customers may impose their favored notation or method or prohibit any with which they are unfamiliar. This factor can conflict with the previous factor.

Note that, in almost all cases, it is useful to start by building a model of the software context. The software context provides a connection between the intended software and its external environment.

This is crucial to understanding the software's context in its operational environment and to identifying its interfaces with the environment.

This subtopic does not seek to "teach" a particular modeling style or notation but rather provides guidance on the purpose and intent of modeling.

4.3. Architectural Design and Requirements Allocation

At some point, the solution architecture must be derived. Architectural design is the point at which the requirements process overlaps with software or systems design and illustrates how impossible it is to cleanly decouple the two tasks. This topic is closely related to Software Structure and Architecture in the Software Design KA. In many cases, the software engineer acts as software architect because the process of analyzing and elaborating the requirements demands that the architecture/design components that will be responsible for satisfying the requirements be identified. This is requirements allocation–the assignment to architecture components responsible for satisfying the requirements.

Allocation is important to permit detailed analysis of requirements. Hence, for example, once a set of requirements has been allocated to a component, the individual requirements can be further analyzed to discover further requirements on how the component needs to interact with other components in order to satisfy the allocated requirements. In large projects, allocation stimulates a new round of analysis for each subsystem. As an example, requirements for a particular braking performance for a car (braking distance, safety in poor driving conditions, smoothness of application, pedal pressure required, and so on) may be allocated to the braking hardware (mechanical and hydraulic assemblies) and an antilock braking system (ABS). Only when a requirement for an antilock braking system has been identified, and the requirements allocated to it, can the capabilities of the ABS, the braking hardware, and emergent properties (such as car weight) be used to identify the detailed ABS software requirements.

Architectural design is closely identified with conceptual modeling (see section 4.2, Conceptual Modeling).

4.4. Requirements Negotiation

Another term commonly used for this subtopic is "conflict resolution." This concerns resolving problems with requirements where conflicts occur between two stakeholders requiring mutually incompatible features, between requirements and resources, or between functional and nonfunctional requirements, for example. In most cases, it is unwise for the software engineer to make a unilateral decision, so it becomes necessary to consult with the stakeholder(s) to reach a consensus on an appropriate tradeoff. It is often important, for contractual reasons, that such decisions be traceable back to the customer. We have classified this as a software requirements analysis topic because problems emerge as the result of analysis. However, a strong case can also be made for considering it a requirements validation topic (see topic 6, Requirements Validation).

Requirements prioritization is necessary, not only as a means to filter important requirements, but also in order to resolve conflicts and plan for staged deliveries, which means making complex decisions that require detailed domain knowledge and good estimation skills. However, it is often difficult to get real information that can act as a basis for such decisions. In addition, requirements often depend on each other, and priorities are relative. In practice, software engineers perform requirements prioritization frequently without knowing about all the requirements. Requirements prioritization may follow a cost-value approach that involves an analysis from the stakeholders defining in a scale the benefits or the aggregated value that the implementation of the requirement brings them, versus the penalties of not having implemented a particular requirement. It also involves an analysis from the software engineers estimating in a scale the cost of implementing each requirement, relative to other requirements. Another requirements prioritization approach called the analytic hierarchy process involves comparing all unique pairs of requirements to determine which of the two is of higher priority, and to what extent.

4.5. Formal Analysis

Formal analysis concerns not only topic 4, but also sections 5.3 and 6.3. This topic is also related to Formal Methods in the Software Engineering Models and Methods Knowledge Area.

Formal analysis has made an impact on some application domains, particularly those of high-integrity systems. The formal expression of requirements requires a language with formally defined semantics. The use of a formal analysis for requirements expression has two benefits. First, it enables requirements expressed in the language to be specified precisely and unambiguously, thus (in principle) avoiding the potential for misinterpretation. Secondly, requirements can be reasoned over, permitting desired properties of the specified software to be proven. Formal reasoning requires tool support to be practicable for anything other than trivial systems, and tools generally fall into two types: theorem provers or model checkers. In neither case can proof be fully automated, and the level of competence in formal reasoning needed in order to use the tools restricts the wider application of formal analysis.

Most formal analysis is focused on relatively late stages of requirements analysis. It is generally counterproductive to apply formalization until the business goals and user requirements have come into sharp focus through means such as those described elsewhere in section 4. However, once the requirements have stabilized and have been elaborated to specify concrete properties of the software, it may be beneficial to formalize at least the critical requirements. This permits static validation that the software specified by the requirements does indeed have the properties (for example, absence of deadlock) that the customer, users, and software engineer expect it to have.

5. Requirements Specification

[1*, c4s2, c4s3, c12s2–5] [2*, c10]

For most engineering professions, the term "specification" refers to the assignment of numerical values or limits to a product's design goals. In software engineering, "software requirements specification" typically refers to the production of a document that can be systematically reviewed, evaluated, and approved. For complex systems, particularly those involving substantial nonsoftware components, as many as three different types of documents are produced: system definition, system requirements, and software requirements. For simple software products, only the third of these is required. All three documents are described here, with the understanding that they may be combined as appropriate. A description of systems engineering can be found in the Related Disciplines of Software Engineering chapter of this *Guide*.

5.1. System Definition Document

This document (sometimes known as the user requirements document or concept of operations document) records the system requirements. It defines the high-level system requirements from the domain perspective. Its readership includes representatives of the system users/customers (marketing may play these roles for market-driven software), so its content must be couched in terms of the domain. The document lists the system requirements along with background information about the overall objectives for the system, its target environment, and a statement of the constraints, assumptions, and nonfunctional requirements. It may include conceptual models designed to illustrate the system context, usage scenarios, and the principal domain entities, as well as workflows.

5.2. System Requirements Specification

Developers of systems with substantial software and nonsoftware components—a modern airliner, for example—often separate the description of system requirements from the description of software requirements. In this view, system requirements are specified, the software requirements are derived from the system requirements, and then the requirements for the software components are specified. Strictly speaking, system requirements specification is a systems engineering activity and falls outside the scope of this *Guide*.

5.3. Software Requirements Specification

Software requirements specification establishes the basis for agreement between customers and contractors or suppliers (in market-driven projects, these roles may be played by the marketing and development divisions) on what the software product is to do as well as what it is not expected to do.

Software requirements specification permits a rigorous assessment of requirements before design can begin and reduces later redesign. It should also provide a realistic basis for estimating product costs, risks, and schedules.

Organizations can also use a software requirements specification document as the basis for developing effective verification and validation plans.

Software requirements specification provides an informed basis for transferring a software product to new users or software platforms. Finally, it can provide a basis for software enhancement.

Software requirements are often written in natural language, but, in software requirements specification, this may be supplemented by formal or semiformal descriptions. Selection of appropriate notations permits particular requirements and aspects of the software architecture to be described more precisely and concisely than natural language. The general rule is that notations should be used that allow the requirements to be described as precisely as possible. This is particularly crucial for safety-critical, regulatory, and certain other types of dependable software. However, the choice of notation is often constrained by the training, skills, and preferences of the document's authors and readers.

A number of quality indicators have been developed that can be used to relate the quality of software requirements specification to other project variables such as cost, acceptance, performance, schedule, and reproducibility. Quality indicators for individual software requirements specification statements include imperatives, directives, weak phrases, options, and continuances. Indicators for the entire software requirements specification document include size, readability, specification, depth, and text structure.

6. Requirements Validation

[1*, c4s6] [2*, c13, c15]

The requirements documents may be subject to validation and verification procedures. The requirements may be validated to ensure that the software engineer has understood the requirements; it is also important to verify that a requirements document conforms to company standards and that it is understandable, consistent, and complete. In cases where documented company standards or terminology are inconsistent with widely accepted standards, a mapping between the two should be agreed on and appended to the document.

Formal notations offer the important advantage of permitting the last two properties to be proven (in a restricted sense, at least). Different stakeholders, including representatives of the customer and developer, should review the document(s). Requirements documents are subject to the same configuration management practices as the other deliverables of the software life cycle processes. When practical, the individual requirements are also subject to configuration management, generally using a requirements management tool (see topic 8, Software Requirements Tools).

It is normal to explicitly schedule one or more points in the requirements process where the requirements are validated. The aim is to pick up any problems before resources are committed to addressing the requirements. Requirements validation is concerned with the process of examining the requirements document to ensure that it defines the right software (that is, the software that the users expect).

6.1. Requirements Reviews

Perhaps the most common means of validation is by inspection or reviews of the requirements document(s). A group of reviewers is assigned a brief to look for errors, mistaken assumptions, lack of clarity, and deviation from standard practice. The composition of the group that conducts the review is important (at least one representative of the customer should be included for a customer-driven project, for example), and it may help to provide guidance on what to look for in the form of checklists.

Reviews may be constituted on completion of the system definition document, the system specification document, the software requirements specification document, the baseline specification for a new release, or at any other step in the process.

6.2. Prototyping

Prototyping is commonly a means for validating the software engineer's interpretation of the software requirements, as well as for eliciting new requirements. As with elicitation, there is a range of prototyping techniques and a number of points in the process where prototype validation may be appropriate. The advantage of prototypes is that they can make it easier to interpret the software engineer's assumptions and, where needed, give useful feedback on why they are wrong. For example, the dynamic behavior of a user interface can be better understood through an animated prototype than through textual description or graphical models. The volatility of a requirement that is defined after prototyping has been done is extremely low because there is agreement between the stakeholder and the software engineer—therefore, for safety-critical and crucial features prototyping would really help. There are also disadvantages, however. These include the danger of users' attention being distracted from the core underlying functionality by cosmetic issues or quality problems with the prototype. For this reason, some advocate prototypes that avoid software, such as flip-chart-based mockups. Prototypes may be costly to develop. However, if they avoid the wastage of resources caused by trying to satisfy erroneous requirements, their cost can be more easily justified. Early prototypes may contain aspects of the final solution. Prototypes may be evolutionary as opposed to throwaway.

6.3. Model Validation

It is typically necessary to validate the quality of the models developed during analysis. For example, in object models, it is useful to perform a static analysis to verify that communication paths exist between objects that, in the stakeholders' domain, exchange data. If formal analysis notations are used, it is possible to use formal reasoning to prove specification properties. This topic is closely related to the Software Engineering Models and Methods KA.

6.4. Acceptance Tests

An essential property of a software requirement is that it should be possible to validate that the finished product satisfies it. Requirements that cannot be validated are really just "wishes." An important task is therefore planning how to verify each requirement. In most cases, designing acceptance tests does this for how end-users typically conduct business using the system.

Identifying and designing acceptance tests may be difficult for nonfunctional requirements (see section 1.3, Functional and Nonfunctional Requirements). To be validated, they must first be analyzed and decomposed to the point where they can be expressed quantitatively.

Additional information can be found in Acceptance/Qualification/Conformance Testing in the Software Testing KA.

7. Practical Considerations

[1*, c4s1, c4s4, c4s6, c4s7]
[2*, c3, c12, c14, c16, c18–21]

The first level of topic decomposition presented in this KA may seem to describe a linear sequence of activities. This is a simplified view of the process.

The requirements process spans the whole software life cycle. Change management and the maintenance of the requirements in a state that accurately mirrors the software to be built, or that has been built, are key to the success of the software engineering process.

Not every organization has a culture of documenting and managing requirements. It is common in dynamic start-up companies, driven by a strong "product vision" and limited resources, to view requirements documentation as unnecessary overhead. Most often, however, as these companies expand, as their customer base grows, and as their product starts to evolve, they discover that they need to recover the requirements that

motivated product features in order to assess the impact of proposed changes. Hence, requirements documentation and change management are key to the success of any requirements process.

7.1. Iterative Nature of the Requirements Process

There is general pressure in the software industry for ever shorter development cycles, and this is particularly pronounced in highly competitive, market-driven sectors. Moreover, most projects are constrained in some way by their environment, and many are upgrades to, or revisions of, existing software where the architecture is a given. In practice, therefore, it is almost always impractical to implement the requirements process as a linear, deterministic process in which software requirements are elicited from the stakeholders, baselined, allocated, and handed over to the software development team. It is certainly a myth that the requirements for large software projects are ever perfectly understood or perfectly specified.

Instead, requirements typically iterate towards a level of quality and detail that is sufficient to permit design and procurement decisions to be made. In some projects, this may result in the requirements being baselined before all their properties are fully understood. This risks expensive rework if problems emerge late in the software engineering process. However, software engineers are necessarily constrained by project management plans and must therefore take steps to ensure that the "quality" of the requirements is as high as possible given the available resources. They should, for example, make explicit any assumptions that underpin the requirements as well as any known problems.

For software products that are developed iteratively, a project team may baseline only those requirements needed for the current iteration. The requirements specialist can continue to develop requirements for future iterations, while developers proceed with design and construction of the current iteration. This approach provides customers with business value quickly, while minimizing the cost of rework.

In almost all cases, requirements understanding continues to evolve as design and development proceeds. This often leads to the revision of requirements late in the life cycle. Perhaps the most crucial point in understanding software requirements is that a significant proportion of the requirements *will* change. This is sometimes due to errors in the analysis, but it is frequently an inevitable consequence of change in the "environment"—for example, the customer's operating or business environment, regulatory processes imposed by the authorities, or the market into which software must sell. Whatever the cause, it is important to recognize the inevitability of change and take steps to mitigate its effects. Change has to be managed by ensuring that proposed changes go through a defined review and approval process and by applying careful requirements tracing, impact analysis, and software configuration management (see the Software Configuration Management KA). Hence, the requirements process is not merely a front-end task in software development, but spans the whole software life cycle. In a typical project, the software requirements activities evolve over time from elicitation to change management. A combination of top-down analysis and design methods and bottom-up implementation and refactoring methods that meet in the middle could provide the best of both worlds. However, this is difficult to achieve in practice, as it depends heavily upon the maturity and expertise of the software engineers.

7.2. Change Management

Change management is central to the management of requirements. This topic describes the role of change management, the procedures that need to be in place, and the analysis that should be applied to proposed changes. It has strong links to the Software Configuration Management KA.

7.3. Requirements Attributes

Requirements should consist not only of a specification of what is required, but also of ancillary information, which helps manage and interpret the requirements. Requirements attributes must be defined, recorded, and updated as the software under development or maintenance evolves. This should include the various classification

dimensions of the requirement (see section 4.1, Requirements Classification) and the verification method or relevant acceptance test plan section. It may also include additional information, such as a summary rationale for each requirement, the source of each requirement, and a change history. The most important requirements attribute, however, is an identifier that allows the requirements to be uniquely and unambiguously identified.

7.4. Requirements Tracing

Requirements tracing is concerned with recovering the source of requirements and predicting the effects of requirements. Tracing is fundamental to performing impact analysis when requirements change. A requirement should be traceable backward to the requirements and stakeholders that motivated it (from a software requirement back to the system requirement(s) that it helps satisfy, for example). Conversely, a requirement should be traceable forward into the requirements and design entities that satisfy it (for example, from a system requirement into the software requirements that have been elaborated from it, and on into the code modules that implement it, or the test cases related to that code and even a given section on the user manual which describes the actual functionality) and into the test case that verifies it.

The requirements tracing for a typical project will form a complex directed acyclic graph (DAG) (see Graphs in the Computing Foundations KA) of requirements. Maintaining an up-to-date graph or traceability matrix is an activity that must be considered during the whole life cycle of a product. If the traceability information is not updated as changes in the requirements continue to happen, the traceability information becomes unreliable for impact analysis.

7.5. Measuring Requirements

As a practical matter, it is typically useful to have some concept of the "volume" of the requirements for a particular software product. This number is useful in evaluating the "size" of a change in requirements, in estimating the cost of a development or maintenance task, or simply for use as the denominator in other measurements. Functional size measurement (FSM) is a technique for evaluating the size of a body of functional requirements.

Additional information on size measurement and standards will be found in the Software Engineering Process KA.

8. Software Requirements Tools

Tools for dealing with software requirements fall broadly into two categories: tools for modeling and tools for managing requirements.

Requirements management tools typically support a range of activities—including documentation, tracing, and change management—and have had a significant impact on practice. Indeed, tracing and change management are really only practicable if supported by a tool. Since requirements management is fundamental to good requirements practice, many organizations have invested in requirements management tools, although many more manage their requirements in more ad hoc and generally less satisfactory ways (e.g., using spreadsheets).

MATRIX OF TOPICS VS. REFERENCE MATERIAL

	Sommerville 2011 [1*]	Wiegers 2003 [2*]
1. Software Requirements Fundamentals		
1.1. Definition of a Software Requirement	c4	c1
1.2. Product and Process Requirements	c4s1	c1, c6
1.3. Functional and Nonfunctional Requirements	c4s1	c12
1.4. Emergent Properties	c10s1	
1.5. Quantifiable Requirements		c1
1.6. System Requirements and Software Requirements	c10s4	c1
2. Requirements Process		
2.1. Process Models	c4s4	c3
2.2. Process Actors		c1, c2, c4, c6
2.3. Process Support and Management		c3
2.4. Process Quality and Improvement		c22, c23
3. Requirements Elicitation		
3.1. Requirements Sources	c4s5	c5, c6,c9
3.2. Elicitation Techniques	c4s5	c6
4. Requirements Analysis		
4.1. Requirements Classification	c4s1	c12
4.2. Conceptual Modeling	c4s5	c11
4.3. Architectural Design and Requirements Allocation	c10s4	c17
4.4. Requirements Negotiation	c4s5	c7
4.5. Formal Analysis	c12s5	
5. Requirements Specification		
5.1. System Definition Document	c4s2	c10
5.2. System Requirements Specification	c4s2, c12s2, c12s3, c12s4, c12s5	c10
5.3. Software Requirements Specification	c4s3	c10
6. Requirements Validation		
6.1. Requirements Reviews	c4s6	c15
6.2. Prototyping	c4s6	c13
6.3. Model Validation	c4s6	c15
6.4. Acceptance Tests	c4s6	c15

	Sommerville 2011 [1*]	Wiegers 2003 [2*]
7. Practical Considerations		
7.1. Iterative Nature of the Requirements Process	c4s4	c3, c16
7.2. Change Management	c4s7	c18, c19
7.3. Requirements Attributes	c4s1	c12, c14
7.4. Requirements Tracing		c20
7.5. Measuring Requirements	c4s6	c18
8. Software Requirements Tools		c21

FURTHER READINGS

I. Alexander and L. Beus-Dukic, *Discovering Requirements* [5].

An easily digestible and practically oriented book on software requirements, this is perhaps the best of current textbooks on how the various elements of software requirements fit together. It is full of practical advice on (for example) how to identify the various system stakeholders and how to evaluate alternative solutions. Its coverage is exemplary and serves as a useful reference for key techniques such as use case modeling and requirements prioritization.

C. Potts, K. Takahashi, and A. Antón, "Inquiry-Based Requirements Analysis" [6].

This paper is an easily digested account of work that has proven to be very influential in the development of requirements handling. It describes how and why the elaboration of requirements cannot be a linear process by which the analyst simply transcribes and reformulates requirements elicited from the customer. The role of scenarios is described in a way that helps to define their use in discovering and describing requirements.

A. van Lamsweerde, *Requirements Engineering: From System Goals to UML Models to Software Specifications* [7].

Serves as a good introduction to requirements engineering but its unique value is as a reference book for the KAOS goal-oriented requirements modelling language. Explains why goal modelling is useful and shows how it can integrate with mainstream modelling techniques using UML.

O. Gotel and A. Finkelstein, "An Analysis of the Requirements Traceability Problem" [8].

This paper is a classic reference work on a key element of requirements management. Based on empirical studies, it sets out the reasons for and the barriers to the effective tracing of requirements. It is essential reading for an understanding of why requirements tracing is an essential element of an effective software process.

N. Maiden and C. Ncube, "Acquiring COTS Software Selection Requirements" [9].

This paper is significant because it recognises explicitly that software products often integrate third-party components. It offers insights into the problems of selecting off-the-shelf software to satisfy requirements: there is usually a mismatch. This challenges some of the assumptions underpinning much of traditional requirements handling, which tends to assume custom software.

REFERENCES

[1*] I. Sommerville, *Software Engineering*, 9th ed., Addison-Wesley, 2011.

[2*] K.E. Wiegers, *Software Requirements*, 2nd ed., Microsoft Press, 2003.

[3] INCOSE, *Systems Engineering Handbook: A Guide for System Life Cycle Processes and Activities*, version 3.2.2, International Council on Systems Engineering, 2012.

[4] S. Friedenthal, A. Moore, and R. Steiner, *A Practical Guide to SysML: The Systems Modeling Language*, 2nd ed., Morgan Kaufmann, 2012.

[5] I. Alexander and L. Beus-Deukic, *Discovering Requirements: How to Specify Products and Services*, Wiley, 2009.

[6] C. Potts, K. Takahashi, and A.I. Antón, "Inquiry-Based Requirements Analysis," *IEEE Software,* vol. 11, no. 2, Mar. 1994, pp. 21–32.

[7] A. van Lamsweerde, *Requirements Engineering: From System Goals to UML Models to Software Specifications*, Wiley, 2009.

[8] O. Gotel and C.W. Finkelstein, "An Analysis of the Requirements Traceability Problem," *Proc. 1st Int'l Conf. Requirements Eng.*, IEEE, 1994.

[9] N.A. Maiden and C. Ncube, "Acquiring COTS Software Selection Requirements," *IEEE Software,* vol. 15, no. 2, Mar.–Apr. 1998, pp. 46–56.

CHAPTER 2

SOFTWARE DESIGN

ACRONYMS

ADL	Architecture Description Language
CBD	Component-Based Design
CRC	Class Responsibility Collaborator
DFD	Data Flow Diagram
ERD	Entity Relationship Diagram
IDL	Interface Description Language
MVC	Model View Controller
OO	Object-Oriented
PDL	Program Design Language

INTRODUCTION

Design is defined as both "the process of defining the architecture, components, interfaces, and other characteristics of a system or component" and "the result of [that] process" [1]. Viewed as a process, software design is the software engineering life cycle activity in which software requirements are analyzed in order to produce a description of the software's internal structure that will serve as the basis for its construction. A software design (the result) describes the software architecture—that is, how software is decomposed and organized into components—and the interfaces between those components. It should also describe the components at a level of detail that enables their construction.

Software design plays an important role in developing software: during software design, software engineers produce various models that form a kind of blueprint of the solution to be implemented. We can analyze and evaluate these models to determine whether or not they will allow us to fulfill the various requirements. We can also examine and evaluate alternative solutions and tradeoffs. Finally, we can use the resulting models to plan subsequent development activities, such as system verification and validation, in addition to using them as inputs and as the starting point of construction and testing.

In a standard list of software life cycle processes, such as that in ISO/IEC/IEEE Std. 12207, *Software Life Cycle Processes* [2], software design consists of two activities that fit between software requirements analysis and software construction:

- Software architectural design (sometimes called high-level design): develops top-level structure and organization of the software and identifies the various components.
- Software detailed design: specifies each component in sufficient detail to facilitate its construction.

This Software Design knowledge area (KA) does not discuss every topic that includes the word "design." In Tom DeMarco's terminology [3], the topics discussed in this KA deal mainly with D-design (decomposition design), the goal of which is to map software into component pieces. However, because of its importance in the field of software architecture, we will also address FP-design (family pattern design), the goal of which is to establish exploitable commonalities in a family of software products. This KA does not address I-design (invention design), which is usually performed during the software requirements process with the goal of conceptualizing and specifying software to satisfy discovered needs and requirements, since this topic is considered to be part of the requirements process (see the Software Requirements KA).

This Software Design KA is related specifically to the Software Requirements, Software

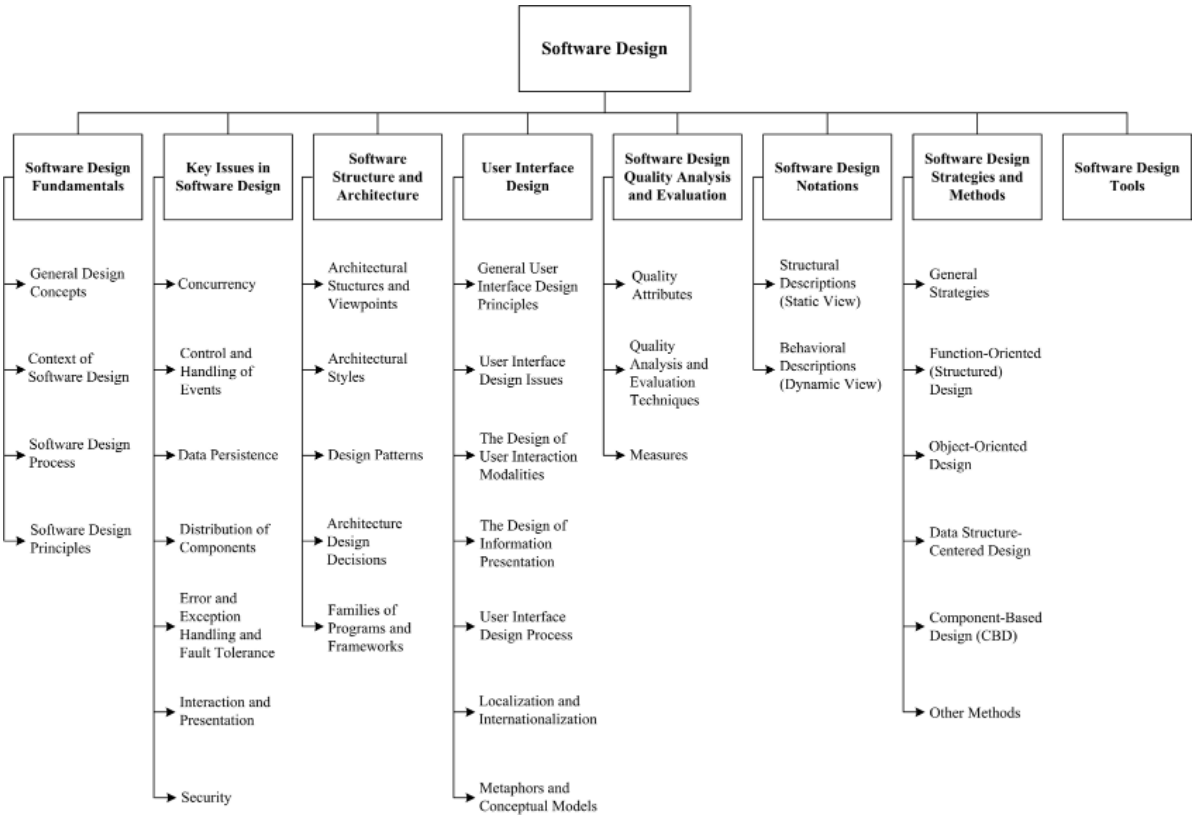

Figure 2.1. Breakdown of Topics for the Software Design KA

Construction, Software Engineering Management, Software Engineering Models and Methods, Software Quality, and Computing Foundations KAs.

BREAKDOWN OF TOPICS FOR SOFTWARE DESIGN

The breakdown of topics for the Software Design KA is shown in Figure 2.1.

1. Software Design Fundamentals

The concepts, notions, and terminology introduced here form an underlying basis for understanding the role and scope of software design.

1.1. General Design Concepts

[4*, c1]

In the general sense, design can be viewed as a form of problem solving. For example, the concept of a wicked problem—a problem with no definitive solution—is interesting in terms of understanding the limits of design. A number of other notions and concepts are also of interest in understanding design in its general sense: goals, constraints, alternatives, representations, and solutions (see Problem Solving Techniques in the Computing Foundations KA).

1.2. Context of Software Design

[4*, c3]

Software design is an important part of the software development process. To understand the role of software design, we must see how it fits in the software development life cycle. Thus, it is important to understand the major characteristics of software requirements analysis, software design, software construction, software testing, and software maintenance.

1.3. Software Design Process

[4*, c2]

Software design is generally considered a two-step process:

- Architectural design (also referred to as high-level design and top-level design) describes how software is organized into components.
- Detailed design describes the desired behavior of these components.

The output of these two processes is a set of models and artifacts that record the major decisions that have been taken, along with an explanation of the rationale for each nontrivial decision. By recording the rationale, long-term maintainability of the software product is enhanced.

1.4. Software Design Principles

[4*] [5*, c6, c7, c21] [6*, c1, c8, c9]

A *principle* is "a comprehensive and fundamental law, doctrine, or assumption" [7]. Software design principles are key notions that provide the basis for many different software design approaches and concepts. Software design principles include abstraction; coupling and cohesion; decomposition and modularization; encapsulation/information hiding; separation of interface and implementation; sufficiency, completeness, and primitiveness; and separation of concerns.

- *Abstraction* is "a view of an object that focuses on the information relevant to a particular purpose and ignores the remainder of the information" [1] (see Abstraction in the Computing Foundations KA). In the context of software design, two key abstraction mechanisms are parameterization and specification. Abstraction by parameterization abstracts from the details of data representations by representing the data as named parameters. Abstraction by specification leads to three major kinds of abstraction: procedural abstraction, data abstraction, and control (iteration) abstraction.
- *Coupling and Cohesion.* Coupling is defined as "a measure of the interdependence among modules in a computer program," whereas cohesion is defined as "a measure of the strength of association of the elements within a module" [1].
- *Decomposition and modularization.* Decomposing and modularizing means that large software is divided into a number of smaller named components having well-defined interfaces that describe component interactions. Usually the goal is to place different functionalities and responsibilities in different components.
- *Encapsulation and information hiding* means grouping and packaging the internal details of an abstraction and making those details inaccessible to external entities.
- *Separation of interface and implementation.* Separating interface and implementation involves defining a component by specifying a public interface (known to the clients) that is separate from the details of how the component is realized (see encapsulation and information hiding above).
- *Sufficiency, completeness, and primitiveness.* Achieving sufficiency and completeness means ensuring that a software component captures all the important characteristics of an abstraction and nothing more. Primitiveness means the design should be based on patterns that are easy to implement.
- *Separation of concerns.* A concern is an "area of interest with respect to a software design" [8]. A design concern is an area of design that is relevant to one or more of its stakeholders. Each architecture view frames one or more concerns. Separating concerns by views allows interested stakeholders to focus on a few things at a time and offers a means of managing complexity [9].

2. Key Issues in Software Design

A number of key issues must be dealt with when designing software. Some are quality concerns that all software must address—for example, performance, security, reliability, usability, etc. Another important issue is how to decompose, organize, and package software components. This is so fundamental that all design approaches address it in one way or another (see section 1.4, Software Design Principles, and topic 7, Software Design Strategies and Methods). In contrast, other issues "deal with some aspect of software's behavior that is not in the application domain, but which addresses some of the supporting

domains" [10]. Such issues, which often crosscut the system's functionality, have been referred to as *aspects*, which "tend not to be units of software's functional decomposition, but rather to be properties that affect the performance or semantics of the components in systemic ways" [11]. A number of these key, crosscutting issues are discussed in the following sections (presented in alphabetical order).

2.1. Concurrency

[5*, c18]

Design for concurrency is concerned with decomposing software into processes, tasks, and threads and dealing with related issues of efficiency, atomicity, synchronization, and scheduling.

2.2. Control and Handling of Events

[5*, c21]

This design issue is concerned with how to organize data and control flow as well as how to handle reactive and temporal events through various mechanisms such as implicit invocation and call-backs.

2.3. Data Persistence

[12*, c9]

This design issue is concerned with how to handle long-lived data.

2.4. Distribution of Components

[5*, c18]

This design issue is concerned with how to distribute the software across the hardware (including computer hardware and network hardware), how the components communicate, and how middleware can be used to deal with heterogeneous software.

2.5. Error and Exception Handling and Fault Tolerance

[5*, c18]

This design issue is concerned with how to prevent, tolerate, and process errors and deal with exceptional conditions.

2.6. Interaction and Presentation

[5*, c16]

This design issue is concerned with how to structure and organize interactions with users as well as the presentation of information (for example, separation of presentation and business logic using the Model-View-Controller approach). Note that this topic does not specify user interface details, which is the task of user interface design (see topic 4, User Interface Design).

2.7. Security

[5*, c12, c18] [13*, c4]

Design for security is concerned with how to prevent unauthorized disclosure, creation, change, deletion, or denial of access to information and other resources. It is also concerned with how to tolerate security-related attacks or violations by limiting damage, continuing service, speeding repair and recovery, and failing and recovering securely. Access control is a fundamental concept of security, and one should also ensure the proper use of cryptology.

3. Software Structure and Architecture

In its strict sense, a software architecture is "the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both" [14*]. During the mid-1990s, however, software architecture started to emerge as a broader discipline that involved the study of software structures and architectures in a more generic way. This gave rise to a number of interesting concepts about software design at different levels of abstraction. Some of these concepts can be useful during the architectural design (for example, architectural styles) as well as during the detailed design (for example, design patterns). These design concepts can also be used to design families of programs (also known as product lines). Interestingly, most of these concepts can be seen as attempts to describe, and thus reuse, design knowledge.

3.1. Architectural Structures and Viewpoints

[14*, c1]

Different high-level facets of a software design can be described and documented. These facets are often called views: "A view represents a partial aspect of a software architecture that shows specific properties of a software system" [14*]. Views pertain to distinct issues associated with software design—for example, the logical view (satisfying the functional requirements) vs. the process view (concurrency issues) vs. the physical view (distribution issues) vs. the development view (how the design is broken down into implementation units with explicit representation of the dependencies among the units). Various authors use different terminologies—like behavioral vs. functional vs. structural vs. data modeling views. In summary, a software design is a multifaceted artifact produced by the design process and generally composed of relatively independent and orthogonal views.

3.2. Architectural Styles

[14*, c1, c2, c3, c4, c5]

An architectural style is "a specialization of element and relation types, together with a set of constraints on how they can be used" [14*]. An architectural style can thus be seen as providing the software's high-level organization. Various authors have identified a number of major architectural styles:

- General structures (for example, layers, pipes and filters, blackboard)
- Distributed systems (for example, client-server, three-tiers, broker)
- Interactive systems (for example, Model-View-Controller, Presentation-Abstraction-Control)
- Adaptable systems (for example, microkernel, reflection)
- Others (for example, batch, interpreters, process control, rule-based).

3.3. Design Patterns

[15*, c3, c4, c5]

Succinctly described, a pattern is "a common solution to a common problem in a given context" [16]. While architectural styles can be viewed as patterns describing the high-level organization of software, other design patterns can be used to describe details at a lower level. These lower level design patterns include the following:

- Creational patterns (for example, builder, factory, prototype, singleton)
- Structural patterns (for example, adapter, bridge, composite, decorator, façade, flyweight, proxy)
- Behavioral patterns (for example, command, interpreter, iterator, mediator, memento, observer, state, strategy, template, visitor).

3.4. Architecture Design Decisions

[5*, c6]

Architectural design is a creative process. During the design process, software designers have to make a number of fundamental decisions that profoundly affect the software and the development process. It is useful to think of the architectural design process from a decision-making perspective rather than from an activity perspective. Often, the impact on quality attributes and tradeoffs among competing quality attributes are the basis for design decisions.

3.5. Families of Programs and Frameworks

[5*, c6, c7, c16]

One approach to providing for reuse of software designs and components is to design families of programs, also known as software product lines. This can be done by identifying the commonalities among members of such families and by designing reusable and customizable components to account for the variability among family members.

In object-oriented (OO) programming, a key related notion is that of a framework: a partially completed software system that can be extended by appropriately instantiating specific extensions (such as plug-ins).

4. User Interface Design

User interface design is an essential part of the software design process. User interface design should ensure that interaction between the human and the machine provides for effective operation

and control of the machine. For software to achieve its full potential, the user interface should be designed to match the skills, experience, and expectations of its anticipated users.

4.1. General User Interface Design Principles

[5*, c29-web] [17*, c2][1]

- *Learnability*. The software should be easy to learn so that the user can rapidly start working with the software.
- *User familiarity*. The interface should use terms and concepts drawn from the experiences of the people who will use the software.
- *Consistency*. The interface should be consistent so that comparable operations are activated in the same way.
- *Minimal surprise.* The behavior of software should not surprise users.
- *Recoverability.* The interface should provide mechanisms allowing users to recover from errors.
- *User guidance.* The interface should give meaningful feedback when errors occur and provide context-related help to users.
- *User diversity*. The interface should provide appropriate interaction mechanisms for diverse types of users and for users with different capabilities (blind, poor eyesight, deaf, colorblind, etc.).

4.2. User Interface Design Issues

[5*, c29-web] [17*, c2]

User interface design should solve two key issues:

- How should the user interact with the software?
- How should information from the software be presented to the user?

User interface design must integrate user interaction and information presentation. User interface design should consider a compromise between the most appropriate styles of interaction and presentation for the software, the background and experience of the software users, and the available devices.

4.3. The Design of User Interaction Modalities

[5*, c29-web] [17*, c2]

User interaction involves issuing commands and providing associated data to the software. User interaction styles can be classified into the following primary styles:

- *Question-answer.* The interaction is essentially restricted to a single question-answer exchange between the user and the software. The user issues a question to the software, and the software returns the answer to the question.
- *Direct manipulation*. Users interact with objects on the computer screen. Direct manipulation often includes a pointing device (such as a mouse, trackball, or a finger on touch screens) that manipulates an object and invokes actions that specify what is to be done with that object.
- *Menu selection*. The user selects a command from a menu list of commands.
- *Form fill-in*. The user fills in the fields of a form. Sometimes fields include menus, in which case the form has action buttons for the user to initiate action.
- *Command language*. The user issues a command and provides related parameters to direct the software what to do.
- *Natural language*. The user issues a command in natural language. That is, the natural language is a front end to a command language and is parsed and translated into software commands.

4.4. The Design of Information Presentation

[5*, c29-web] [17*, c2]

Information presentation may be textual or graphical in nature. A good design keeps the information presentation separate from the information itself. The MVC (Model-View-Controller) approach is an effective way to keep information presentation separating from the information being presented.

1 Chapter 29 is a web-based chapter available at http://ifs.host.cs.st-andrews.ac.uk/Books/SE9/WebChapters/.

Software engineers also consider software response time and feedback in the design of information presentation. Response time is generally measured from the point at which a user executes a certain control action until the software responds with a response. An indication of progress is desirable while the software is preparing the response. Feedback can be provided by restating the user's input while processing is being completed.

Abstract visualizations can be used when large amounts of information are to be presented.

According to the style of information presentation, designers can also use color to enhance the interface. There are several important guidelines:

- Limit the number of colors used.
- Use color change to show the change of software status.
- Use color-coding to support the user's task.
- Use color-coding in a thoughtful and consistent way.
- Use colors to facilitate access for people with color blindness or color deficiency (e.g., use the change of color saturation and color brightness, try to avoid blue and red combinations).
- Don't depend on color alone to convey important information to users with different capabilities (blindness, poor eyesight, color-blindness, etc.).

4.5. User Interface Design Process

[5*, c29-web] [17*, c2]

User interface design is an iterative process; interface prototypes are often used to determine the features, organization, and look of the software user interface. This process includes three core activities:

- *User analysis.* In this phase, the designer analyzes the users' tasks, the working environment, other software, and how users interact with other people.
- *Software prototyping.* Developing prototype software help users to guide the evolution of the interface.
- *Interface evaluation.* Designers can observe users' experiences with the evolving interface.

4.6. Localization and Internationalization

[17*, c8, c9]

User interface design often needs to consider internationalization and localization, which are means of adapting software to the different languages, regional differences, and the technical requirements of a target market. Internationalization is the process of designing a software application so that it can be adapted to various languages and regions without major engineering changes. Localization is the process of adapting internationalized software for a specific region or language by adding locale-specific components and translating the text. Localization and internationalization should consider factors such as symbols, numbers, currency, time, and measurement units.

4.7. Metaphors and Conceptual Models

[17*, c5]

User interface designers can use metaphors and conceptual models to set up mappings between the software and some reference system known to the users in the real world, which can help the users to more readily learn and use the interface. For example, the operation "delete file" can be made into a metaphor using the icon of a trash can.

When designing a user interface, software engineers should be careful to not use more than one metaphor for each concept. Metaphors also present potential problems with respect to internationalization, since not all metaphors are meaningful or are applied in the same way within all cultures.

5. Software Design Quality Analysis and Evaluation

This section includes a number of quality analysis and evaluation topics that are specifically related to software design. (See also the Software Quality KA.)

5.1. Quality Attributes

[4*, c4]

Various attributes contribute to the quality of a software design, including various "-ilities" (maintainability, portability, testability, usability)

and "-nesses" (correctness, robustness). There is an interesting distinction between quality attributes discernible at runtime (for example, performance, security, availability, functionality, usability), those not discernible at runtime (for example, modifiability, portability, reusability, testability), and those related to the architecture's intrinsic qualities (for example, conceptual integrity, correctness, completeness). (See also the Software Quality KA.)

5.2. Quality Analysis and Evaluation Techniques
[4*, c4] [5*, c24]

Various tools and techniques can help in analyzing and evaluating software design quality.

- Software design reviews: informal and formalized techniques to determine the quality of design artifacts (for example, architecture reviews, design reviews, and inspections; scenario-based techniques; requirements tracing). Software design reviews can also evaluate security. Aids for installation, operation, and usage (for example, manuals and help files) can be reviewed.
- Static analysis: formal or semiformal static (nonexecutable) analysis that can be used to evaluate a design (for example, fault-tree analysis or automated cross-checking). Design vulnerability analysis (for example, static analysis for security weaknesses) can be performed if security is a concern. Formal design analysis uses mathematical models that allow designers to predicate the behavior and validate the performance of the software instead of having to rely entirely on testing. Formal design analysis can be used to detect residual specification and design errors (perhaps caused by imprecision, ambiguity, and sometimes other kinds of mistakes). (See also the Software Engineering Models and Methods KA.)
- Simulation and prototyping: dynamic techniques to evaluate a design (for example, performance simulation or feasibility prototypes).

5.3. Measures
[4*, c4] [5*, c24]

Measures can be used to assess or to quantitatively estimate various aspects of a software design; for example, size, structure, or quality. Most measures that have been proposed depend on the approach used for producing the design. These measures are classified in two broad categories:

- Function-based (structured) design measures: measures obtained by analyzing functional decomposition; generally represented using a structure chart (sometimes called a hierarchical diagram) on which various measures can be computed.
- Object-oriented design measures: the design structure is typically represented as a class diagram, on which various measures can be computed. Measures on the properties of the internal content of each class can also be computed.

6. Software Design Notations

Many notations exist to represent software design artifacts. Some are used to describe the structural organization of a design, others to represent software behavior. Certain notations are used mostly during architectural design and others mainly during detailed design, although some notations can be used for both purposes. In addition, some notations are used mostly in the context of specific design methods (see topic 7, Software Design Strategies and Methods). Please note that software design is often accomplished using multiple notations. Here, they are categorized into notations for describing the structural (static) view vs. the behavioral (dynamic) view.

6.1. Structural Descriptions (Static View)
[4*, c7] [5*, c6, c7] [6*, c4, c5, c6, c7] [12*, c7] [14*, c7]

The following notations, mostly but not always graphical, describe and represent the structural aspects of a software design—that is, they are

used to describe the major components and how they are interconnected (static view):

- Architecture description languages (ADLs): textual, often formal, languages used to describe software architecture in terms of components and connectors.
- Class and object diagrams: used to represent a set of classes (and objects) and their interrelationships.
- Component diagrams: used to represent a set of components ("physical and replaceable part[s] of a system that [conform] to and [provide] the realization of a set of interfaces" [18]) and their interrelationships.
- Class responsibility collaborator cards (CRCs): used to denote the names of components (class), their responsibilities, and their collaborating components' names.
- Deployment diagrams: used to represent a set of (physical) nodes and their interrelationships, and, thus, to model the physical aspects of software.
- Entity-relationship diagrams (ERDs): used to represent conceptual models of data stored in information repositories.
- Interface description languages (IDLs): programming-like languages used to define the interfaces (names and types of exported operations) of software components.
- Structure charts: used to describe the calling structure of programs (which modules call, and are called by, which other modules).

6.2. Behavioral Descriptions (Dynamic View)
[4*, c7, c13] [5*, c6, c7] [6*, c4, c5, c6, c7] [14*, c8]

The following notations and languages, some graphical and some textual, are used to describe the dynamic behavior of software systems and components. Many of these notations are useful mostly, but not exclusively, during detailed design. Moreover, behavioral descriptions can include a rationale for design decision such as how a design will meet security requirements.

- Activity diagrams: used to show control flow from activity to activity. Can be used to represent concurrent activities.
- Communication diagrams: used to show the interactions that occur among a group of objects; emphasis is on the objects, their links, and the messages they exchange on those links.
- Data flow diagrams (DFDs): used to show data flow among elements. A data flow diagram provides "a description based on modeling the flow of information around a network of operational elements, with each element making use of or modifying the information flowing into that element" [4*]. Data flows (and therefore data flow diagrams) can be used for security analysis, as they offer identification of possible paths for attack and disclosure of confidential information.
- Decision tables and diagrams: used to represent complex combinations of conditions and actions.
- Flowcharts: used to represent the flow of control and the associated actions to be performed.
- Sequence diagrams: used to show the interactions among a group of objects, with emphasis on the time ordering of messages passed between objects.
- State transition and state chart diagrams: used to show the control flow from state to state and how the behavior of a component changes based on its current state in a state machine.
- Formal specification languages: textual languages that use basic notions from mathematics (for example, logic, set, sequence) to rigorously and abstractly define software component interfaces and behavior, often in terms of pre- and postconditions. (See also the Software Engineering Models and Methods KA.)
- Pseudo code and program design languages (PDLs): structured programming-like languages used to describe, generally at the detailed design stage, the behavior of a procedure or method.

7. Software Design Strategies and Methods

There exist various general strategies to help guide the design process. In contrast with general strategies, methods are more specific in that they generally provide a set of notations to be used with the method, a description of the process to be used when following the method, and a set of guidelines for using the method. Such methods are useful as a common framework for teams of software engineers. (See also the Software Engineering Models and Methods KA).

7.1. General Strategies

[4*, c8, c9, c10] [12*, c7]

Some often-cited examples of general strategies useful in the design process include the divide-and-conquer and stepwise refinement strategies, top-down vs. bottom-up strategies, and strategies making use of heuristics, use of patterns and pattern languages, and use of an iterative and incremental approach.

7.2. Function-Oriented (Structured) Design

[4*, c13]

This is one of the classical methods of software design, where decomposition centers on identifying the major software functions and then elaborating and refining them in a hierarchical top-down manner. Structured design is generally used after structured analysis, thus producing (among other things) data flow diagrams and associated process descriptions. Researchers have proposed various strategies (for example, transformation analysis, transaction analysis) and heuristics (for example, fan-in/fan-out, scope of effect vs. scope of control) to transform a DFD into a software architecture generally represented as a structure chart.

7.3. Object-Oriented Design

[4*, c16]

Numerous software design methods based on objects have been proposed. The field has evolved from the early object-oriented (OO) design of the mid-1980s (noun = object; verb = method; adjective = attribute), where inheritance and polymorphism play a key role, to the field of component-based design, where metainformation can be defined and accessed (through reflection, for example). Although OO design's roots stem from the concept of data abstraction, responsibility-driven design has been proposed as an alternative approach to OO design.

7.4. Data Structure-Centered Design

[4*, c14, c15]

Data structure-centered design starts from the data structures a program manipulates rather than from the function it performs. The software engineer first describes the input and output data structures and then develops the program's control structure based on these data structure diagrams. Various heuristics have been proposed to deal with special cases—for example, when there is a mismatch between the input and output structures.

7.5. Component-Based Design (CBD)

[4*, c17]

A software component is an independent unit, having well-defined interfaces and dependencies that can be composed and deployed independently. Component-based design addresses issues related to providing, developing, and integrating such components in order to improve reuse. Reused and off-the-shelf software components should meet the same security requirements as new software. Trust management is a design concern; components treated as having a certain degree of trustworthiness should not depend on less trustworthy components or services.

7.6. Other Methods

[5*, c19, c21]

Other interesting approaches also exist (see the Software Engineering Models and Methods KA). Iterative and adaptive methods implement software increments and reduce emphasis on rigorous software requirement and design.

Aspect-oriented design is a method by which software is constructed using aspects to implement the crosscutting concerns and extensions that are identified during the software requirements process. Service-oriented architecture is a way to build distributed software using web services executed on distributed computers. Software systems are often constructed by using services from different providers because standard protocols (such as HTTP, HTTPS, SOAP) have been designed to support service communication and service information exchange.

8. Software Design Tools

[14*, c10, Appendix A]

Software design tools can be used to support the creation of the software design artifacts during the software development process. They can support part or whole of the following activities:

- to translate the requirements model into a design representation;
- to provide support for representing functional components and their interface(s);
- to implement heuristics refinement and partitioning;
- to provide guidelines for quality assessment.

MATRIX OF TOPICS VS. REFERENCE MATERIAL

	Budgen 2003 [4*]	Sommerville 2011 [5*]	Page-Jones 1999 [6*]	Brookshear 2008 [12*]	Allen 2008 [13*]	Clements et al. 2010 [14*]	Gamma et al. 1994 [15*]	Nielsen 1993 [17*]
1. Software Design Fundamentals								
1.1. General Design Concepts	c1							
1.2. The Context of Software Design	c3							
1.3. The Software Design Process	c2							
1.4. Software Design Principles	c1	c6, c7, c21	c1, c8, c9					
2. Key Issues in Software Design								
2.1. Concurrency		c18						
2.2. Control and Handling of Events		c21						
2.3. Data Persistence				c9				
2.4. Distribution of Components		c18						
2.5. Error and Exception Handling and Fault Tolerance		c18						
2.6. Interaction and Presentation		c16						
2.7. Security		c12, c18			c4			
3. Software Structure and Architecture								
3.1. Architectural Structures and Viewpoints						c1		
3.2. Architectural Styles						c1, c2, c3, c4, c5		
3.3. Design Patterns							c3, c4, c5	

	Budgen 2003 [4*]	Sommerville 2011 [5*]	Page-Jones 1999 [6*]	Brookshear 2008 [12*]	Allen 2008 [13*]	Clements et al. 2010 [14*]	Gamma et al. 1994 [15*]	Nielsen 1993 [17*]
3.4. Architecture Design Decisions		c6						
3.5. Families of Programs and Frameworks		c6, c7, c16						
4. User Interface Design								
4.1. General User Interface Design Principle		c29-web						c2
4.2. User Interface Design Issues		c29-web						
4.3. The Design of User Interaction Modalities		c29-web						
4.4. The Design of Information Presentation		c29-web						
4.5. User Interface Design Process		c29-web						
4.6. Localization and Internationalization								c8, c9
4.7. Metaphors and Conceptual Models								c5
5. Software Design Quality Analysis and Evaluation								
5.1. Quality Attributes	c4							
5.2. Quality Analysis and Evaluation Techniques	c4	c24						
5.3. Measures	c4	c24						

	Budgen 2003 [4*]	Sommerville 2011 [5*]	Page-Jones 1999 [6*]	Brookshear 2008 [12*]	Allen 2008 [13*]	Clements et al. 2010 [14*]	Gamma et al. 1994 [15*]	Nielsen 1993 [17*]
6. Software Design Notations								
6.1. Structural Descriptions (Static View)	c7	c6, c7	c4, c5, c6, c7	c7		c7		
6.2. Behavioral Descriptions (Dynamic View)	c7, c13, c18	c6, c7	c4, c5, c6, c7			c8		
7. Software Design Strategies and Methods								
7.1. General Strategies	c8, c9, c10			c7				
7.2. Function-Oriented (Structured) Design	c13							
7.3. Object-Oriented Design	c16							
7.4. Data Structure-Centered Design	c14, c15							
7.5. Component-Based Design (CBD)	c17							
7.6. Other Methods		c19, c21						
8. Software Design Tools						c10, App. A		

FURTHER READINGS

Roger Pressman, *Software Engineering: A Practitioner's Approach (Seventh Edition)* [19].

For roughly three decades, Roger Pressman's *Software Engineering: A Practitioner's Approach* has been one of the world's leading textbooks in software engineering. Notably, this complementary textbook to [5*] comprehensively presents software design—including design concepts, architectural design, component-level design, user interface design, pattern-based design, and web application design.

"The 4+1 View Model of Architecture" [20].

The seminal paper "The 4+1 View Model" organizes a description of a software architecture using five concurrent views. The four views of the model are the logical view, the development view, the process view, and the physical view. In addition, selected use cases or scenarios are utilized to illustrate the architecture. Hence, the model contains 4+1 views. The views are used to describe the software as envisioned by different stakeholders—such as end-users, developers, and project managers.

Len Bass, Paul Clements, and Rick Kazman, *Software Architecture in Practice* [21].

This book introduces the concepts and best practices of software architecture, meaning how software is structured and how the software's components interact. Drawing on their own experience, the authors cover the essential technical topics for designing, specifying, and validating software architectures. They also emphasize the importance of the business context in which large software is designed. Their aim is to present software architecture in a real-world setting, reflecting both the opportunities and constraints that organizations encounter. This is one of the best books currently available on software architecture.

REFERENCES

[1] *ISO/IEC/IEEE 24765:2010 Systems and Software Engineering—Vocabulary*, ISO/IEC/IEEE, 2010.

[2] *IEEE Std. 12207-2008 (a.k.a. ISO/IEC 12207:2008) Standard for Systems and Software Engineering—Software Life Cycle Processes*, IEEE, 2008.

[3] T. DeMarco, "The Paradox of Software Architecture and Design," Stevens Prize Lecture, 1999.

[4*] D. Budgen, *Software Design*, 2nd ed., Addison-Wesley, 2003.

[5*] I. Sommerville, *Software Engineering*, 9th ed., Addison-Wesley, 2011.

[6*] M. Page-Jones, *Fundamentals of Object-Oriented Design in UML*, 1st ed., Addison-Wesley, 1999.

[7] *Merriam-Webster's Collegiate Dictionary*, 11th ed., 2003.

[8] *IEEE Std. 1069-2009 Standard for Information Technology—Systems Design—Software Design Descriptions*, IEEE, 2009.

[9] *ISO/IEC 42010:2011 Systems and Software Engineering—Recommended Practice for Architectural Description of Software-Intensive Systems*, ISO/IEC, 2011.

[10] J. Bosch, *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*, ACM Press, 2000.

[11] G. Kiczales et al., "Aspect-Oriented Programming," *Proc. 11th European Conf. Object-Oriented Programming* (ECOOP 97), Springer, 1997.

[12*] J.G. Brookshear, *Computer Science: An Overview*, 10th ed., Addison-Wesley, 2008.

[13*] J.H. Allen et al., *Software Security Engineering: A Guide for Project Managers*, Addison-Wesley, 2008.

[14*] P. Clements et al., *Documenting Software Architectures: Views and Beyond*, 2nd ed., Pearson Education, 2010.

[15*] E. Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed., Addison-Wesley Professional, 1994.

[16] I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process*, Addison-Wesley Professional, 1999.

[17*] J. Nielsen, *Usability Engineering*, Morgan Kaufmann, 1993.

[18] G. Booch, J. Rumbaugh, and I. Jacobson, The Unified Modeling Language User Guide, Addison-Wesley, 1999.

[19] R.S. Pressman, *Software Engineering: A Practitioner's Approach*, 7th ed., McGraw-Hill, 2010.

[20] P.B. Kruchten, "The 4+1 View Model of Architecture," *IEEE Software,* vol. 12, no. 6, 1995, pp. 42–55.

[21] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd ed., Addison-Wesley Professional, 2013.

CHAPTER 3

SOFTWARE CONSTRUCTION

ACRONYMS

API	Application Programming Interface
COTS	Commercial Off-the-Shelf
GUI	Graphical User Interface
IDE	Integrated Development Environment
OMG	Object Management Group
POSIX	Portable Operating System Interface
TDD	Test-Driven Development
UML	Unified Modeling Language

INTRODUCTION

The term software construction refers to the detailed creation of working software through a combination of coding, verification, unit testing, integration testing, and debugging.

The Software Construction knowledge area (KA) is linked to all the other KAs, but it is most strongly linked to Software Design and Software Testing because the software construction process involves significant software design and testing. The process uses the design output and provides an input to testing ("design" and "testing" in this case referring to the activities, not the KAs). Boundaries between design, construction, and testing (if any) will vary depending on the software life cycle processes that are used in a project.

Although some detailed design may be performed prior to construction, much design work is performed during the construction activity. Thus, the Software Construction KA is closely linked to the Software Design KA.

Throughout construction, software engineers both unit test and integration test their work. Thus, the Software Construction KA is closely linked to the Software Testing KA as well.

Software construction typically produces the highest number of configuration items that need to be managed in a software project (source files, documentation, test cases, and so on). Thus, the Software Construction KA is also closely linked to the Software Configuration Management KA.

While software quality is important in all the KAs, code is the ultimate deliverable of a software project, and thus the Software Quality KA is closely linked to the Software Construction KA.

Since software construction requires knowledge of algorithms and of coding practices, it is closely related to the Computing Foundations KA, which is concerned with the computer science foundations that support the design and construction of software products. It is also related to project management, insofar as the management of construction can present considerable challenges.

BREAKDOWN OF TOPICS FOR SOFTWARE CONSTRUCTION

Figure 3.1 gives a graphical representation of the top-level decomposition of the breakdown for the Software Construction KA.

1. Software Construction Fundamentals

Software construction fundamentals include

- minimizing complexity
- anticipating change
- constructing for verification
- reuse
- standards in construction.

The first four concepts apply to design as well as to construction. The following sections define

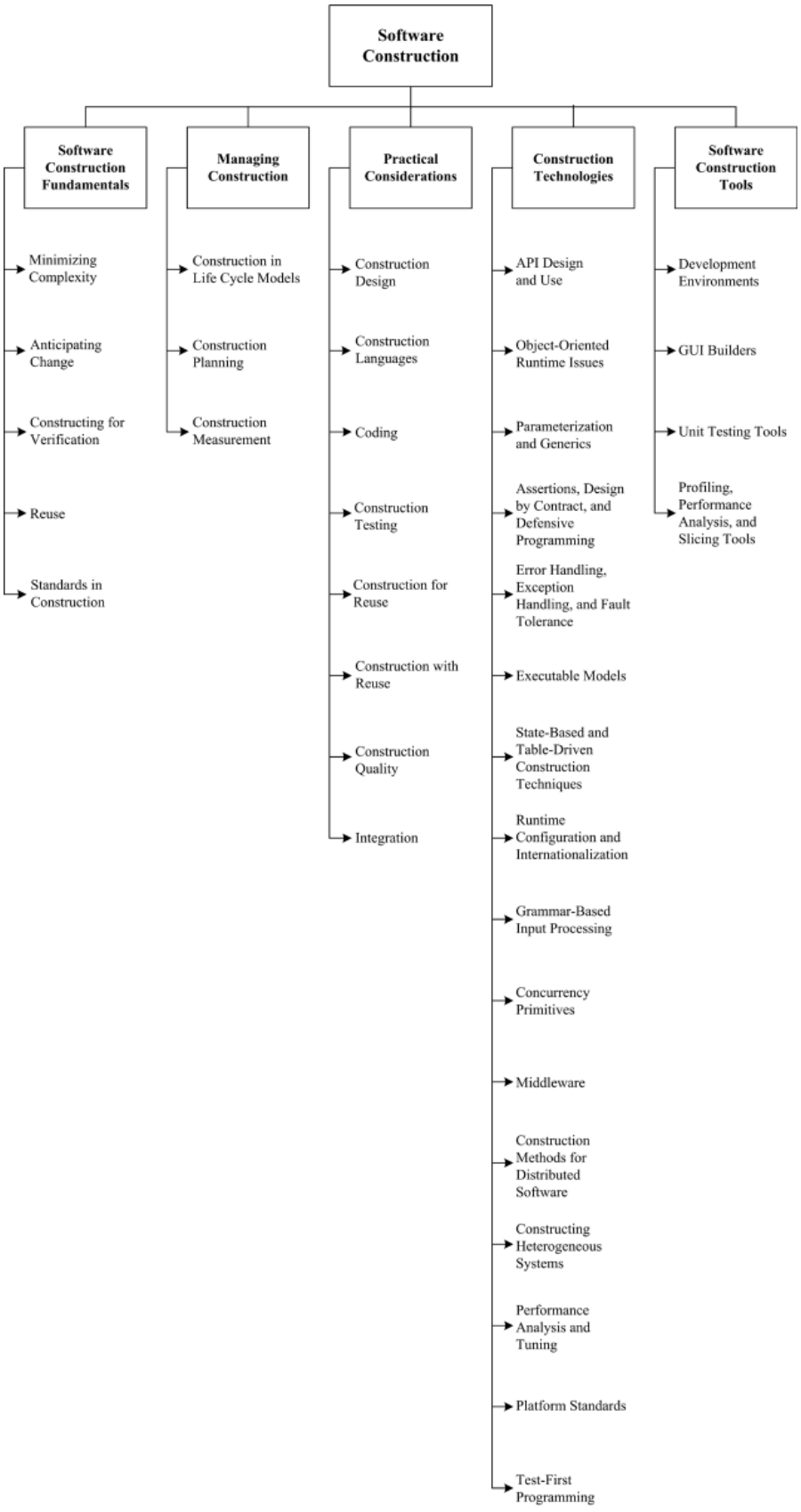

Figure 3.1. Breakdown of Topics for the Software Construction KA

these concepts and describe how they apply to construction.

1.1. Minimizing Complexity

[1*]

Most people are limited in their ability to hold complex structures and information in their working memories, especially over long periods of time. This proves to be a major factor influencing how people convey intent to computers and leads to one of the strongest drives in software construction: *minimizing* complexity. The need to reduce complexity applies to essentially every aspect of software construction and is particularly critical to testing of software constructions.

In software construction, reduced complexity is achieved through emphasizing code creation that is simple and readable rather than clever. It is accomplished through making use of standards (see section 1.5, Standards in Construction), modular design (see section 3.1, Construction Design), and numerous other specific techniques (see section 3.3, Coding). It is also supported by construction-focused quality techniques (see section 3.7, Construction Quality).

1.2. Anticipating Change

[1*]

Most software will change over time, and the anticipation of *change* drives many aspects of software construction; changes in the environments in which software operates also affect software in diverse ways.

Anticipating change helps software engineers build extensible software, which means they can enhance a software product without disrupting the underlying structure.

Anticipating change is supported by many specific techniques (see section 3.3, Coding).

1.3. Constructing for Verification

[1*]

Constructing for verification means building software in such a way that faults can be readily found by the software engineers writing the software as well as by the testers and users during independent testing and operational activities. Specific techniques that support constructing for verification include following coding standards to support code reviews and unit testing, organizing code to support automated testing, and restricting the use of complex or hard-to-understand language structures, among others.

1.4. Reuse

[2*]

Reuse refers to using existing assets in solving different problems. In software construction, typical assets that are reused include libraries, modules, components, source code, and commercial off-the-shelf (COTS) assets. Reuse is best practiced systematically, according to a well-defined, repeatable process. Systematic reuse can enable significant software productivity, quality, and cost improvements.

Reuse has two closely related facets: "construction for reuse" and "construction with reuse." The former means to create reusable software assets, while the latter means to reuse software assets in the construction of a new solution. Reuse often transcends the boundary of projects, which means reused assets can be constructed in other projects or organizations.

1.5. Standards in Construction

[1*]

Applying external or internal development standards during construction helps achieve a project's objectives for efficiency, quality, and cost. Specifically, the choices of allowable programming language subsets and usage standards are important aids in achieving higher security.

Standards that directly affect construction issues include

- communication methods (for example, standards for document formats and contents)
- programming languages (for example, language standards for languages like Java and C++)
- coding standards (for example, standards for naming conventions, layout, and indentation)
- platforms (for example, interface standards for operating system calls)

- tools (for example, diagrammatic standards for notations like UML (Unified Modeling Language)).

Use of external standards. Construction depends on the use of external standards for construction languages, construction tools, technical interfaces, and interactions between the Software Construction KA and other KAs. Standards come from numerous sources, including hardware and software interface specifications (such as the Object Management Group (OMG)) and international organizations (such as the IEEE or ISO).

Use of internal standards. Standards may also be created on an organizational basis at the corporate level or for use on specific projects. These standards support coordination of group activities, minimizing complexity, anticipating change, and constructing for verification.

2. Managing Construction

2.1. Construction in Life Cycle Models

[1*]

Numerous models have been created to develop software; some emphasize construction more than others.

Some models are more linear from the construction point of view—such as the waterfall and staged-delivery life cycle models. These models treat construction as an activity that occurs only after significant prerequisite work has been completed—including detailed requirements work, extensive design work, and detailed planning. The more linear approaches tend to emphasize the activities that precede construction (requirements and design) and to create more distinct separations between activities. In these models, the main emphasis of construction may be coding.

Other models are more iterative—such as evolutionary prototyping and agile development. These approaches tend to treat construction as an activity that occurs concurrently with other software development activities (including requirements, design, and planning) or that overlaps them. These approaches tend to mix design, coding, and testing activities, and they often treat the combination of activities as construction (see the Software Management and Software Process KAs).

Consequently, what is considered to be "construction" depends to some degree on the life cycle model used. In general, software construction is mostly coding and debugging, but it also involves construction planning, detailed design, unit testing, integration testing, and other activities.

2.2. Construction Planning

[1*]

The choice of construction method is a key aspect of the construction-planning activity. The choice of construction method affects the extent to which construction prerequisites are performed, the order in which they are performed, and the degree to which they should be completed before construction work begins.

The approach to construction affects the project team's ability to reduce complexity, anticipate change, and construct for verification. Each of these objectives may also be addressed at the process, requirements, and design levels—but they will be influenced by the choice of construction method.

Construction planning also defines the order in which components are created and integrated, the integration strategy (for example, phased or incremental integration), the software quality management processes, the allocation of task assignments to specific software engineers, and other tasks, according to the chosen method.

2.3. Construction Measurement

[1*]

Numerous construction activities and artifacts can be measured—including code developed, code modified, code reused, code destroyed, code complexity, code inspection statistics, fault-fix and fault-find rates, effort, and scheduling. These measurements can be useful for purposes of managing construction, ensuring quality during construction, and improving the construction process, among other uses (see the Software Engineering Process KA for more on measurement).

3. Practical Considerations

Construction is an activity in which the software engineer has to deal with sometimes chaotic and changing real-world constraints, and he or she must do so precisely. Due to the influence of real-world constraints, construction is more driven by practical considerations than some other KAs, and software engineering is perhaps most craft-like in the construction activities.

3.1. Construction Design

[1*]

Some projects allocate considerable design activity to construction, while others allocate design to a phase explicitly focused on design. Regardless of the exact allocation, some detailed design work will occur at the construction level, and that design work tends to be dictated by constraints imposed by the real-world problem that is being addressed by the software.

Just as construction workers building a physical structure must make small-scale modifications to account for unanticipated gaps in the builder's plans, software construction workers must make modifications on a smaller or larger scale to flesh out details of the software design during construction.

The details of the design activity at the construction level are essentially the same as described in the Software Design KA, but they are applied on a smaller scale of algorithms, data structures, and interfaces.

3.2. Construction Languages

[1*]

Construction languages include all forms of communication by which a human can specify an executable problem solution to a problem. Construction languages and their implementations (for example, compilers) can affect software quality attributes of performance, reliability, portability, and so forth. They can be serious contributors to security vulnerabilities.

The simplest type of construction language is a *configuration language,* in which software engineers choose from a limited set of predefined options to create new or custom software installations. The text-based configuration files used in both the Windows and Unix operating systems are examples of this, and the menu-style selection lists of some program generators constitute another example of a configuration language.

Toolkit languages are used to build applications out of elements in toolkits (integrated sets of application-specific reusable parts); they are more complex than configuration languages. Toolkit languages may be explicitly defined as application programming languages, or the applications may simply be implied by a toolkit's set of interfaces.

Scripting languages are commonly used kinds of application programming languages. In some scripting languages, scripts are called batch files or macros.

Programming languages are the most flexible type of construction languages. They also contain the least amount of information about specific application areas and development processes—therefore, they require the most training and skill to use effectively. The choice of programming language can have a large effect on the likelihood of vulnerabilities being introduced during coding—for example, uncritical usage of C and C++ are questionable choices from a security viewpoint.

There are three general kinds of notation used for programming languages, namely

- linguistic (e.g., C/C++, Java)
- formal (e.g., Event-B)
- visual (e.g., MatLab).

Linguistic notations are distinguished in particular by the use of textual strings to represent complex software constructions. The combination of textual strings into patterns may have a sentence-like syntax. Properly used, each such string should have a strong semantic connotation providing an immediate intuitive understanding of what will happen when the software construction is executed.

Formal notations rely less on intuitive, everyday meanings of words and text strings and more on definitions backed up by precise, unambiguous, and formal (or mathematical) definitions. Formal construction notations and formal methods are at the semantic base of most forms of

system programming notations, where accuracy, time behavior, and testability are more important than ease of mapping into natural language. Formal constructions also use precisely defined ways of combining symbols that avoid the ambiguity of many natural language constructions.

Visual notations rely much less on the textual notations of linguistic and formal construction and instead rely on direct visual interpretation and placement of visual entities that represent the underlying software. Visual construction tends to be somewhat limited by the difficulty of making "complex" statements using only the arrangement of icons on a display. However, these icons can be powerful tools in cases where the primary programming task is simply to build and "adjust" a visual interface to a program, the detailed behavior of which has an underlying definition.

3.3. Coding

[1*]

The following considerations apply to the software construction coding activity:

- Techniques for creating understandable source code, including naming conventions and source code layout;
- Use of classes, enumerated types, variables, named constants, and other similar entities;
- Use of control structures;
- Handling of error conditions—both anticipated and exceptional (input of bad data, for example);
- Prevention of code-level security breaches (buffer overflows or array index bounds, for example);
- Resource usage via use of exclusion mechanisms and discipline in accessing serially reusable resources (including threads and database locks);
- Source code organization (into statements, routines, classes, packages, or other structures);
- Code documentation;
- Code tuning,

3.4. Construction Testing

[1*]

Construction involves two forms of testing, which are often performed by the software engineer who wrote the code:

- Unit testing
- Integration testing.

The purpose of construction testing is to reduce the gap between the time when faults are inserted into the code and the time when those faults are detected, thereby reducing the cost incurred to fix them. In some instances, test cases are written after code has been written. In other instances, test cases may be created before code is written.

Construction testing typically involves a subset of the various types of testing, which are described in the Software Testing KA. For instance, construction testing does not typically include system testing, alpha testing, beta testing, stress testing, configuration testing, usability testing, or other more specialized kinds of testing.

Two standards have been published on the topic of construction testing: IEEE Standard 829-1998, *IEEE Standard for Software Test Documentation,* and IEEE Standard 1008-1987, *IEEE Standard for Software Unit Testing*.

(See sections 2.1.1., Unit Testing, and 2.1.2., Integration Testing, in the Software Testing KA for more specialized reference material.)

3.5. Construction for Reuse

[2*]

Construction for reuse creates software that has the potential to be reused in the future for the present project or other projects taking a broad-based, multisystem perspective. Construction for reuse is usually based on variability analysis and design. To avoid the problem of code clones, it is desired to encapsulate reusable code fragments into well-structured libraries or components.

The tasks related to software construction for reuse during coding and testing are as follows:

- Variability implementation with mechanisms such as parameterization, conditional compilation, design patterns, and so forth.
- Variability encapsulation to make the software assets easy to configure and customize.
- Testing the variability provided by the reusable software assets.
- Description and publication of reusable software assets.

3.6. Construction with Reuse

[2*]

Construction with reuse means to create new software with the reuse of existing software assets. The most popular method of reuse is to reuse code from the libraries provided by the language, platform, tools being used, or an organizational repository. Asides from these, the applications developed today widely make use of many open-source libraries. Reused and off-the-shelf software often have the same—or better—quality requirements as newly developed software (for example, security level).

The tasks related to software construction with reuse during coding and testing are as follows:

- The selection of the reusable units, databases, test procedures, or test data.
- The evaluation of code or test reusability.
- The integration of reusable software assets into the current software.
- The reporting of reuse information on new code, test procedures, or test data.

3.7. Construction Quality

[1*]

In addition to faults resulting from requirements and design, faults introduced during construction can result in serious quality problems—for example, security vulnerabilities. This includes not only faults in security functionality but also faults elsewhere that allow bypassing of this functionality and other security weaknesses or violations.

Numerous techniques exist to ensure the quality of code as it is constructed. The primary techniques used for construction quality include

- unit testing and integration testing (see section 3.4, Construction Testing)
- test-first development (see section 2.2 in the Software Testing KA)
- use of assertions and defensive programming
- debugging
- inspections
- technical reviews, including security-oriented reviews (see section 2.3.2 in the Software Quality KA)
- static analysis (see section 2.3 of the Software Quality KA)

The specific technique or techniques selected depend on the nature of the software being constructed as well as on the skillset of the software engineers performing the construction activities. Programmers should know good practices and common vulnerabilities—for example, from widely recognized lists about common vulnerabilities. Automated static analysis of code for security weaknesses is available for several common programming languages and can be used in security-critical projects.

Construction quality activities are differentiated from other quality activities by their focus. Construction quality activities focus on code and artifacts that are closely related to code—such as detailed design—as opposed to other artifacts that are less directly connected to the code, such as requirements, high-level designs, and plans.

3.8. Integration

[1*]

A key activity during construction is the integration of individually constructed routines, classes, components, and subsystems into a single system. In addition, a particular software system may need to be integrated with other software or hardware systems.

Concerns related to construction integration include planning the sequence in which components will be integrated, identifying what hardware is needed, creating scaffolding to support interim versions of the software, determining the degree of testing and quality work performed on components before they are integrated, and

determining points in the project at which interim versions of the software are tested.

Programs can be integrated by means of either the phased or the incremental approach. Phased integration, also called "big bang" integration, entails delaying the integration of component software parts until all parts intended for release in a version are complete. Incremental integration is thought to offer many advantages over the traditional phased integration—for example, easier error location, improved progress monitoring, earlier product delivery, and improved customer relations. In incremental integration, the developers write and test a program in small pieces and then combine the pieces one at a time. Additional test infrastructure, such as stubs, drivers, and mock objects, are usually needed to enable incremental integration. By building and integrating one unit at a time (for example, a class or component), the construction process can provide early feedback to developers and customers. Other advantages of incremental integration include easier error location, improved progress monitoring, more fully tested units, and so forth.

4. Construction Technologies

4.1. API Design and Use

[3*]

An application programming interface (API) is the set of signatures that are exported and available to the users of a library or a framework to write their applications. Besides signatures, an API should always include statements about the program's effects and/or behaviors (i.e., its semantics).

API design should try to make the API easy to learn and memorize, lead to readable code, be hard to misuse, be easy to extend, be complete, and maintain backward compatibility. As the APIs usually outlast their implementations for a widely used library or framework, it is desired that the API be straightforward and kept stable to facilitate the development and maintenance of the client applications.

API use involves the processes of selecting, learning, testing, integrating, and possibly extending APIs provided by a library or framework (see section 3.6, Construction with Reuse).

4.2. Object-Oriented Runtime Issues

[1*]

Object-oriented languages support a series of runtime mechanisms including polymorphism and reflection. These runtime mechanisms increase the flexibility and adaptability of object-oriented programs. Polymorphism is the ability of a language to support general operations without knowing until runtime what kind of concrete objects the software will include. Because the program does not know the exact types of the objects in advance, the exact behaviour is determined at runtime (called dynamic binding).

Reflection is the ability of a program to observe and modify its own structure and behavior at runtime. Reflection allows inspection of classes, interfaces, fields, and methods at runtime without knowing their names at compile time. It also allows instantiation at runtime of new objects and invocation of methods using parameterized class and method names.

4.3. Parameterization and Generics

[4*]

Parameterized types, also known as generics (Ada, Eiffel) and templates (C++), enable the definition of a type or class without specifying all the other types it uses. The unspecified types are supplied as parameters at the point of use. Parameterized types provide a third way (in addition to class inheritance and object composition) to compose behaviors in object-oriented software.

4.4. Assertions, Design by Contract, and Defensive Programming

[1*]

An assertion is an executable predicate that's placed in a program—usually a routine or macro—that allows runtime checks of the program. Assertions are especially useful in high-reliability programs. They enable programmers to more quickly flush out mismatched interface assumptions, errors that creep in when code is modified, and so on. Assertions are normally compiled into the code at development time and are later compiled out of the code so that they don't degrade the performance.

Design by contract is a development approach in which preconditions and postconditions are included for each routine. When preconditions and postconditions are used, each routine or class is said to form a contract with the rest of the program. Furthermore, a contract provides a precise specification of the semantics of a routine, and thus helps the understanding of its behavior. Design by contract is thought to improve the quality of software construction.

Defensive programming means to protect a routine from being broken by invalid inputs. Common ways to handle invalid inputs include checking the values of all the input parameters and deciding how to handle bad inputs. Assertions are often used in defensive programming to check input values.

4.5. Error Handling, Exception Handling, and Fault Tolerance

[1*]

The way that errors are handled affects software's ability to meet requirements related to correctness, robustness, and other nonfunctional attributes. Assertions are sometimes used to check for errors. Other error handling techniques—such as returning a neutral value, substituting the next piece of valid data, logging a warning message, returning an error code, or shutting down the software—are also used.

Exceptions are used to detect and process errors or exceptional events. The basic structure of an exception is that a routine uses *throw* to throw a detected exception and an exception handling block will *catch* the exception in a *try-catch* block. The try-catch block may process the erroneous condition in the routine or it may return control to the calling routine. Exception handling policies should be carefully designed following common principles such as including in the exception message all information that led to the exception, avoiding empty catch blocks, knowing the exceptions the library code throws, perhaps building a centralized exception reporter, and standardizing the program's use of exceptions.

Fault tolerance is a collection of techniques that increase software reliability by detecting errors and then recovering from them if possible or containing their effects if recovery is not possible. The most common fault tolerance strategies include backing up and retrying, using auxiliary code, using voting algorithms, and replacing an erroneous value with a phony value that will have a benign effect.

4.6. Executable Models

[5*]

Executable models abstract away the details of specific programming languages and decisions about the organization of the software. Different from traditional software models, a specification built in an executable modeling language like xUML (executable UML) can be deployed in various software environments without change. An executable-model compiler (transformer) can turn an executable model into an implementation using a set of decisions about the target hardware and software environment. Thus, constructing executable models can be regarded as a way of constructing executable software.

Executable models are one foundation supporting the Model-Driven Architecture (MDA) initiative of the Object Management Group (OMG). An executable model is a way to completely specify a Platform Independent Model (PIM); a PIM is a model of a solution to a problem that does not rely on any implementation technologies. Then a Platform Specific Model (PSM), which is a model that contains the details of the implementation, can be produced by weaving together the PIM and the platform on which it relies.

4.7. State-Based and Table-Driven Construction Techniques

[1*]

State-based programming, or automata-based programming, is a programming technology using finite state machines to describe program behaviours. The transition graphs of a state machine are used in all stages of software development (specification, implementation, debugging, and documentation). The main idea is to construct computer programs the same way the automation of technological processes is done. State-based programming is usually combined

with object-oriented programming, forming a new composite approach called *state-based, object-oriented programming.*

A table-driven method is a schema that uses tables to look up information rather than using logic statements (such as *if* and *case*). Used in appropriate circumstances, table-driven code is simpler than complicated logic and easier to modify. When using table-driven methods, the programmer addresses two issues: what information to store in the table or tables, and how to efficiently access information in the table.

4.8. Runtime Configuration and Internationalization

[1*]

To achieve more flexibility, a program is often constructed to support late binding time of its variables. Runtime configuration is a technique that binds variable values and program settings when the program is running, usually by updating and reading configuration files in a just-in-time mode.

Internationalization is the technical activity of preparing a program, usually interactive software, to support multiple locales. The corresponding activity, *localization,* is the activity of modifying a program to support a specific local language. Interactive software may contain dozens or hundreds of prompts, status displays, help messages, error messages, and so on. The design and construction processes should accommodate string and character-set issues including which character set is to be used, what kinds of strings are used, how to maintain the strings without changing the code, and translating the strings into different languages with minimal impact on the processing code and the user interface.

4.9. Grammar-Based Input Processing

[1*] [6*]

Grammar-based input processing involves syntax analysis, or *parsing*, of the input token stream. It involves the creation of a data structure (called a *parse tree* or *syntax tree*) representing the input data. The inorder traversal of the parse tree usually gives the expression just parsed. The parser checks the symbol table for the presence of programmer-defined variables that populate the tree. After building the parse tree, the program uses it as input to the computational processes.

4.10. Concurrency Primitives

[7*]

A synchronization primitive is a programming abstraction provided by a programming language or the operating system that facilitates concurrency and synchronization. Well-known concurrency primitives include semaphores, monitors, and mutexes.

A semaphore is a protected variable or abstract data type that provides a simple but useful abstraction for controlling access to a common resource by multiple processes or threads in a concurrent programming environment.

A monitor is an abstract data type that presents a set of programmer-defined operations that are executed with mutual exclusion. A monitor contains the declaration of shared variables and procedures or functions that operate on those variables. The monitor construct ensures that only one process at a time is active within the monitor.

A mutex (mutual exclusion) is a synchronization primitive that grants exclusive access to a shared resource by only one process or thread at a time.

4.11. Middleware

[3*] [6*]

Middleware is a broad classification for software that provides services above the operating system layer yet below the application program layer. Middleware can provide runtime containers for software components to provide message passing, persistence, and a transparent location across a network. Middleware can be viewed as a connector between the components that use the middleware. Modern message-oriented middleware usually provides an Enterprise Service Bus (ESB), which supports service-oriented interaction and communication between multiple software applications.

4.12. Construction Methods for Distributed Software

[7*]

A distributed system is a collection of physically separate, possibly heterogeneous computer systems that are networked to provide the users with access to the various resources that the system maintains. Construction of distributed software is distinguished from traditional software construction by issues such as parallelism, communication, and fault tolerance.

Distributed programming typically falls into one of several basic architectural categories: client-server, 3-tier architecture, n-tier architecture, distributed objects, loose coupling, or tight coupling (see section 14.3 of the Computing Foundations KA and section 3.2 of the Software Design KA).

4.13. Constructing Heterogeneous Systems

[6*]

Heterogeneous systems consist of a variety of specialized computational units of different types, such as Digital Signal Processors (DSPs), microcontrollers, and peripheral processors. These computational units are independently controlled and communicate with one another. Embedded systems are typically heterogeneous systems.

The design of heterogeneous systems may require the combination of several specification languages in order to design different parts of the system—in other words, hardware/software codesign. The key issues include multilanguage validation, cosimulation, and interfacing.

During the hardware/software codesign, software development and virtual hardware development proceed concurrently through stepwise decomposition. The hardware part is usually simulated in field programmable gate arrays (FPGAs) or application-specific integrated circuits (ASICs). The software part is translated into a low-level programming language.

4.14. Performance Analysis and Tuning

[1*]

Code efficiency—determined by architecture, detailed design decisions, and data-structure and algorithm selection—influences an execution speed and size. Performance analysis is the investigation of a program's behavior using information gathered as the program executes, with the goal of identifying possible hot spots in the program to be improved.

Code tuning, which improves performance at the code level, is the practice of modifying correct code in ways that make it run more efficiently. Code tuning usually involves only small-scale changes that affect a single class, a single routine, or, more commonly, a few lines of code. A rich set of code tuning techniques is available, including those for tuning logic expressions, loops, data transformations, expressions, and routines. Using a low-level language is another common technique for improving some hot spots in a program.

4.15. Platform Standards

[6*] [7*]

Platform standards enable programmers to develop portable applications that can be executed in compatible environments without changes. Platform standards usually involve a set of standard services and APIs that compatible platform implementations must implement. Typical examples of platform standards are Java 2 Platform Enterprise Edition (J2EE) and the POSIX standard for operating systems (Portable Operating System Interface), which represents a set of standards implemented primarily for UNIX-based operating systems.

4.16. Test-First Programming

[1*]

Test-first programming (also known as Test-Driven Development—TDD) is a popular development style in which test cases are written prior to writing any code. Test-first programming can usually detect defects earlier and correct them more easily than traditional programming styles. Furthermore, writing test cases first forces programmers to think about requirements and design before coding, thus exposing requirements and design problems sooner.

5. Software Construction Tools

5.1. Development Environments

[1*]

A development environment, or integrated development environment (IDE), provides comprehensive facilities to programmers for software construction by integrating a set of development tools. The choices of development environments can affect the efficiency and quality of software construction.

In additional to basic code editing functions, modern IDEs often offer other features like compilation and error detection from within the editor, integration with source code control, build/test/debugging tools, compressed or outline views of programs, automated code transforms, and support for refactoring.

5.2. GUI Builders

[1*]

A GUI (Graphical User Interface) builder is a software development tool that enables the developer to create and maintain GUIs in a WYSIWYG (what you see is what you get) mode. A GUI builder usually includes a visual editor for the developer to design forms and windows and manage the layout of the widgets by dragging, dropping, and parameter setting. Some GUI builders can automatically generate the source code corresponding to the visual GUI design.

Because current GUI applications usually follow the event-driven style (in which the flow of the program is determined by events and event handling), GUI builder tools usually provide code generation assistants, which automate the most repetitive tasks required for event handling. The supporting code connects widgets with the outgoing and incoming events that trigger the functions providing the application logic.

Some modern IDEs provide integrated GUI builders or GUI builder plug-ins. There are also many standalone GUI builders.

5.3. Unit Testing Tools

[1*] [2*]

Unit testing verifies the functioning of software modules in isolation from other software elements that are separately testable (for example, classes, routines, components). Unit testing is often automated. Developers can use unit testing tools and frameworks to extend and create automated testing environment. With unit testing tools and frameworks, the developer can code criteria into the test to verify the unit's correctness under various data sets. Each individual test is implemented as an object, and a test runner runs all of the tests. During the test execution, those failed test cases will be automatically flagged and reported.

5.4. Profiling, Performance Analysis, and Slicing Tools

[1*]

Performance analysis tools are usually used to support code tuning. The most common performance analysis tools are profiling tools. An execution profiling tool monitors the code while it runs and records how many times each statement is executed or how much time the program spends on each statement or execution path. Profiling the code while it is running gives insight into how the program works, where the hot spots are, and where the developers should focus the code tuning efforts.

Program slicing involves computation of the set of program statements (i.e., the program slice) that may affect the values of specified variables at some point of interest, which is referred to as a slicing criterion. Program slicing can be used for locating the source of errors, program understanding, and optimization analysis. Program slicing tools compute program slices for various programming languages using static or dynamic analysis methods.

MATRIX OF TOPICS VS. REFERENCE MATERIAL

	McConnell 2004 [1*]	Sommerville 2011 [2*]	Clements et al. 2010 [3*]	Gamma et al. 1994 [4*]	Mellor and Balcer 2002 [5*]	Null and Lobur 2006 [6*]	Silberschatz et al. 2008 [7*]
1. Software Construction Fundamentals							
1.1. Minimizing Complexity	c2, c3, c7-c9, c24, c27, c28, c31, c32, c34						
1.2. Anticipating Change	c3–c5, c24, c31, c32, c34						
1.3. Constructing for Verification	c8, c20–c23, c31, c34						
1.4. Reuse		c16					
1.5. Standards in Construction	c4						
2. Managing Construction							
2.1. Construction in Life Cycle Models	c2, c3, c27, c29						
2.2. Construction Planning	c3, c4, c21, c27–c29						
2.3. Construction Measurement	c25, c28						
3. Practical Considerations							
3.1. Construction Design	c3, c5, c24						
3.2. Construction Languages	c4						
3.3. Coding	c5–c19, c25–c26						

	McConnell 2004 [1*]	Sommerville 2011 [2*]	Clements et al. 2010 [3*]	Gamma et al. 1994 [4*]	Mellor and Balcer 2002 [5*]	Null and Lobur 2006 [6*]	Silberschatz et al. 2008 [7*]
3.4. Construction Testing	c22, c23						
3.5. Construction for Reuse		c16					
3.6. Construction with Reuse		c16					
3.7. Construction Quality	c8, c20–c25						
3.8. Integration	c29						
4. Construction Technologies							
4.1. API Design and Use			c7				
4.2. Object-Oriented Runtime Issues	c6, c7						
4.3. Parameterization and Generics				c1			
4.4. Assertions, Design by Contract, and Defensive Programming	c8, c9						
4.5. Error Handling, Exception Handling, and Fault Tolerance	c3, c8						
4.6. Executable Models					c1		
4.7. State-Based and Table-Driven Construction Techniques	c18						
4.8. Runtime Configuration and Internationalization	c3, c10						
4.9. Grammar-Based Input Processing	c5					c8	

	McConnell 2004 [1*]	Sommerville 2011 [2*]	Clements et al. 2010 [3*]	Gamma et al. 1994 [4*]	Mellor and Balcer 2002 [5*]	Null and Lobur 2006 [6*]	Silberschatz et al. 2008 [7*]
4.10. Concurrency Primitives							c6
4.11. Middleware			c1			c8	
4.12. Construction Methods for Distributed Software							c2
4.13. Constructing Heterogeneous Systems						c9	
4.14. Performance Analysis and Tuning	c25, c26						
4.15. Platform Standards						c10	c1
4.16. Test-First Programming	c22						
5. Construction Tools							
5.1. Development Environments	c30						
5.2. GUI Builders	c30						
5.3. Unit Testing Tools	c22	c8					
5.4. Profiling, Performance Analysis, and Slicing Tools	c25, c26						

FURTHER READINGS

IEEE Std. 1517-2010 Standard for Information Technology—System and Software Life Cycle Processes—Reuse Processes, IEEE, 2010 [8].

This standard specifies the processes, activities, and tasks to be applied during each phase of the software life cycle to enable a software product to be constructed from reusable assets. It covers the concept of reuse-based development and the processes of construction for reuse and construction with reuse.

IEEE Std. 12207-2008 (a.k.a. ISO/IEC 12207:2008) Standard for Systems and Software Engineering—Software Life Cycle Processes, IEEE, 2008 [9].

This standard defines a series of software development processes, including software construction process, software integration process, and software reuse process.

REFERENCES

[1*] S. McConnell, *Code Complete*, 2nd ed., Microsoft Press, 2004.

[2*] I. Sommerville, *Software Engineering*, 9th ed., Addison-Wesley, 2011.

[3*] P. Clements et al., *Documenting Software Architectures: Views and Beyond*, 2nd ed., Pearson Education, 2010.

[4*] E. Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed., Addison-Wesley Professional, 1994.

[5*] S.J. Mellor and M.J. Balcer, *Executable UML: A Foundation for Model-Driven Architecture*, 1st ed., Addison-Wesley, 2002.

[6*] L. Null and J. Lobur, *The Essentials of Computer Organization and Architecture*, 2nd ed., Jones and Bartlett Publishers, 2006.

[7*] A. Silberschatz, P.B. Galvin, and G. Gagne, *Operating System Concepts*, 8th ed., Wiley, 2008.

[8] *IEEE Std. 1517-2010 Standard for Information Technology—System and Software Life Cycle Processes—Reuse Processes*, IEEE, 2010.

[9] *IEEE Std. 12207-2008 (a.k.a. ISO/IEC 12207:2008) Standard for Systems and Software Engineering—Software Life Cycle Processes*, IEEE, 2008.

CHAPTER 4

SOFTWARE TESTING

ACRONYMS

API	Application Program Interface
TDD	Test-Driven Development
TTCN3	Testing and Test Control Notation Version 3
XP	Extreme Programming

INTRODUCTION

Software testing consists of the *dynamic* verification that a program provides *expected* behaviors on a *finite* set of test cases, suitably *selected* from the usually infinite execution domain.

In the above definition, italicized words correspond to key issues in describing the Software Testing knowledge area (KA):

- *Dynamic:* This term means that testing always implies executing the program on selected inputs. To be precise, the input value alone is not always sufficient to specify a test, since a complex, nondeterministic system might react to the same input with different behaviors, depending on the system state. In this KA, however, the term "input" will be maintained, with the implied convention that its meaning also includes a specified input state in those cases for which it is important. Static techniques are different from and complementary to dynamic testing. Static techniques are covered in the Software Quality KA. It is worth noting that terminology is not uniform among different communities and some use the term "testing" also in reference to static techniques.
- *Finite:* Even in simple programs, so many test cases are theoretically possible that exhaustive testing could require months or years to execute. This is why, in practice, a complete set of tests can generally be considered infinite, and testing is conducted on a subset of all possible tests, which is determined by risk and prioritization criteria. Testing always implies a tradeoff between limited resources and schedules on the one hand and inherently unlimited test requirements on the other.
- *Selected:* The many proposed test techniques differ essentially in how the test set is selected, and software engineers must be aware that different selection criteria may yield vastly different degrees of effectiveness. How to identify the most suitable selection criterion under given conditions is a complex problem; in practice, risk analysis techniques and software engineering expertise are applied.
- *Expected:* It must be possible, although not always easy, to decide whether the observed outcomes of program testing are acceptable or not; otherwise, the testing effort is useless. The observed behavior may be checked against user needs (commonly referred to as testing for validation), against a specification (testing for verification), or, perhaps, against the anticipated behavior from implicit requirements or expectations (see Acceptance Tests in the Software Requirements KA).

In recent years, the view of software testing has matured into a constructive one. Testing is no longer seen as an activity that starts only after the coding phase is complete with the limited purpose of detecting failures. Software testing is, or should be, pervasive throughout the entire development and maintenance life cycle. Indeed, planning for software testing should start with the early stages of the software requirements process,

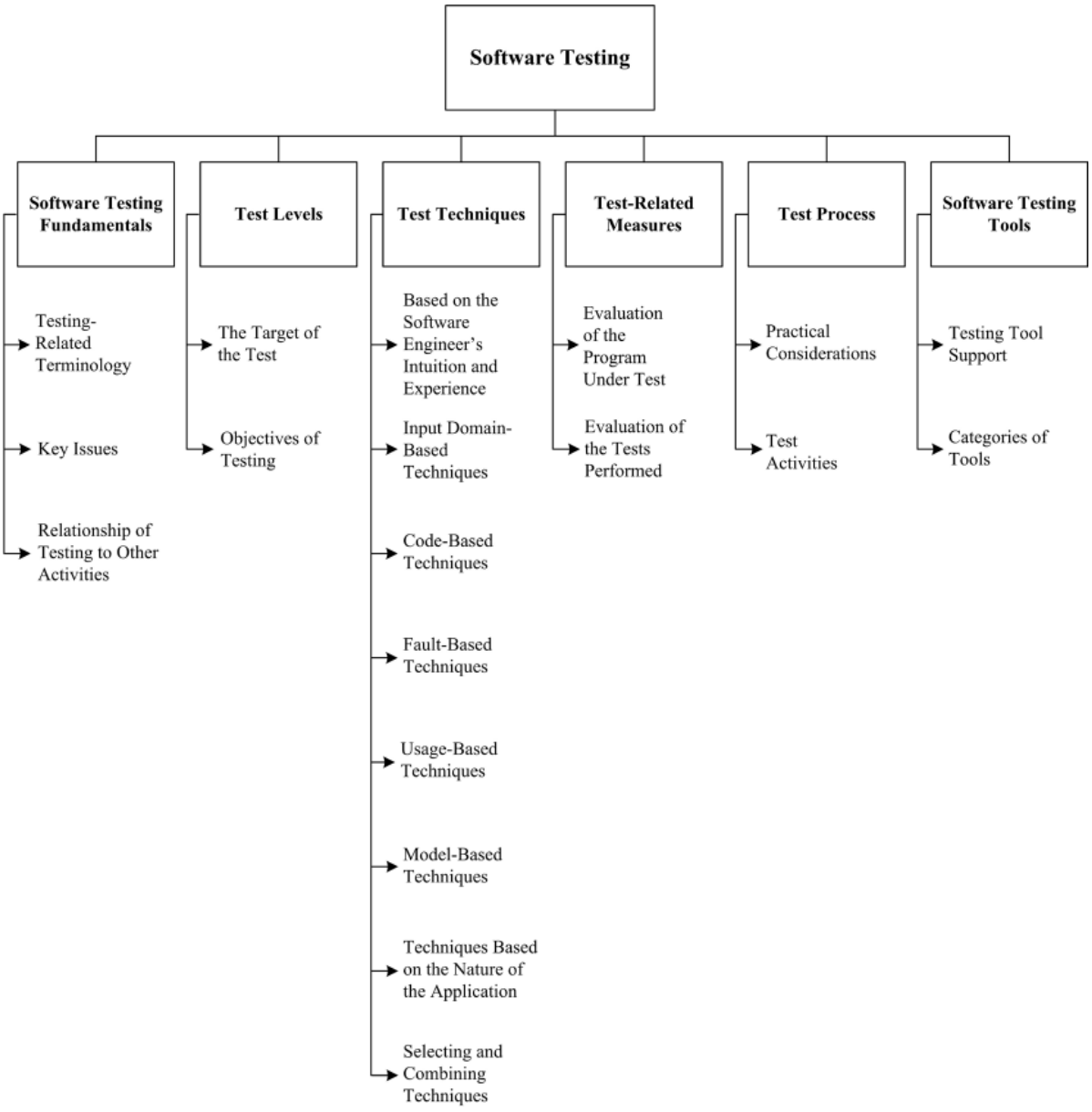

Figure 4.1. Breakdown of Topics for the Software Testing KA

and test plans and procedures should be systematically and continuously developed—and possibly refined—as software development proceeds. These test planning and test designing activities provide useful input for software designers and help to highlight potential weaknesses, such as design oversights/contradictions, or omissions/ambiguities in the documentation.

For many organizations, the approach to software quality is one of prevention: it is obviously much better to prevent problems than to correct them. Testing can be seen, then, as a means for providing information about the functionality and quality attributes of the software and also for identifying faults in those cases where error prevention has not been effective. It is perhaps obvious but worth recognizing that software can still contain faults, even after completion of an extensive testing activity. Software failures experienced after delivery are addressed by corrective maintenance. Software maintenance topics are covered in the Software Maintenance KA.

In the Software Quality KA (see Software Quality Management Techniques), software quality management techniques are notably categorized into static techniques (no code execution) and

dynamic techniques (code execution). Both categories are useful. This KA focuses on dynamic techniques.

Software testing is also related to software construction (see Construction Testing in the Software Construction KA). In particular, unit and integration testing are intimately related to software construction, if not part of it.

BREAKDOWN OF TOPICS FOR SOFTWARE TESTING

The breakdown of topics for the Software Testing KA is shown in Figure 4.1. A more detailed breakdown is provided in the Matrix of Topics vs. Reference Material at the end of this KA.

The first topic describes Software Testing Fundamentals. It covers the basic definitions in the field of software testing, the basic terminology and key issues, and software testing's relationship with other activities.

The second topic, Test Levels, consists of two (orthogonal) subtopics: the first subtopic lists the levels in which the testing of large software is traditionally subdivided, and the second subtopic considers testing for specific conditions or properties and is referred to as Objectives of Testing. Not all types of testing apply to every software product, nor has every possible type been listed.

The test target and test objective together determine how the test set is identified, both with regard to its consistency—*how much testing is enough for achieving the stated objective*—and to its composition—*which test cases should be selected for achieving the stated objective* (although usually "for achieving the stated objective" remains implicit and only the first part of the two italicized questions above is posed). Criteria for addressing the first question are referred to as *test adequacy criteria*, while those addressing the second question are the test *selection criteria*.

Several Test Techniques have been developed in the past few decades, and new ones are still being proposed. Generally accepted techniques are covered in the third topic.

Test-Related Measures are dealt with in the fourth topic, while the issues relative to Test Process are covered in the fifth. Finally, Software Testing Tools are presented in topic six.

1. Software Testing Fundamentals

1.1. Testing-Related Terminology

1.1.1. Definitions of Testing and Related Terminology

[1*, c1, c2] [2*, c8]

Definitions of testing and testing-related terminology are provided in the cited references and summarized as follows.

1.1.2. Faults vs. Failures

[1*, c1s5] [2*, c11]

Many terms are used in the software engineering literature to describe a malfunction: notably *fault*, *failure*, and *error,* among others. This terminology is precisely defined in [3, c2]. It is essential to clearly distinguish between the *cause* of a malfunction (for which the term fault will be used here) and an undesired effect observed in the system's delivered service (which will be called a failure). Indeed there may well be faults in the software that never manifest themselves as failures (see Theoretical and Practical Limitations of Testing in section 1.2, Key Issues). Thus testing can reveal failures, but it is the faults that can and must be removed [3]. The more generic term *defect* can be used to refer to either a fault or a failure, when the distinction is not important [3].

However, it should be recognized that the cause of a failure cannot always be unequivocally identified. No theoretical criteria exist to definitively determine, in general, the fault that caused an observed failure. It might be said that it was the fault that had to be modified to remove the failure, but other modifications might have worked just as well. To avoid ambiguity, one could refer to *failure-causing inputs* instead of faults—that is, those sets of inputs that cause a failure to appear.

1.2. Key Issues

1.2.1. Test Selection Criteria / Test Adequacy Criteria (Stopping Rules)

[1*, c1s14, c6s6, c12s7]

A test selection criterion is a means of selecting test cases or determining that a set of test cases

is sufficient for a specified purpose. Test adequacy criteria can be used to decide when sufficient testing will be, or has been accomplished [4] (see Termination in section 5.1, Practical Considerations).

1.2.2. Testing Effectiveness / Objectives for Testing

[1*, c11s4, c13s11]

Testing effectiveness is determined by analyzing a set of program executions. Selection of tests to be executed can be guided by different objectives: it is only in light of the objective pursued that the effectiveness of the test set can be evaluated.

1.2.3. Testing for Defect Discovery

[1*, c1s14]

In testing for defect discovery, a successful test is one that causes the system to fail. This is quite different from testing to demonstrate that the software meets its specifications or other desired properties, in which case testing is successful if no failures are observed under realistic test cases and test environments.

1.2.4. The Oracle Problem

[1*, c1s9, c9s7]

An oracle is any human or mechanical agent that decides whether a program behaved correctly in a given test and accordingly results in a verdict of "pass" or "fail." There exist many different kinds of oracles; for example, unambiguous requirements specifications, behavioral models, and code annotations. Automation of mechanized oracles can be difficult and expensive.

1.2.5. Theoretical and Practical Limitations of Testing

[1*, c2s7]

Testing theory warns against ascribing an unjustified level of confidence to a series of successful tests. Unfortunately, most established results of testing theory are negative ones, in that they state what testing can never achieve as opposed to what is actually achieved. The most famous quotation in this regard is the Dijkstra aphorism that "program testing can be used to show the presence of bugs, but never to show their absence" [5]. The obvious reason for this is that complete testing is not feasible in realistic software. Because of this, testing must be driven based on risk [6, part 1] and can be seen as a risk management strategy.

1.2.6. The Problem of Infeasible Paths

[1*, c4s7]

Infeasible paths are control flow paths that cannot be exercised by any input data. They are a significant problem in path-based testing, particularly in automated derivation of test inputs to exercise control flow paths.

1.2.7. Testability

[1*, c17s2]

The term "software testability" has two related but different meanings: on the one hand, it refers to the ease with which a given test coverage criterion can be satisfied; on the other hand, it is defined as the likelihood, possibly measured statistically, that a set of test cases will expose a failure *if* the software is faulty. Both meanings are important.

1.3. Relationship of Testing to Other Activities

Software testing is related to, but different from, static software quality management techniques, proofs of correctness, debugging, and program construction. However, it is informative to consider testing from the point of view of software quality analysts and of certifiers.

- Testing vs. Static Software Quality Management Techniques (see Software Quality Management Techniques in the Software Quality KA [1*, c12]).
- Testing vs. Correctness Proofs and Formal Verification (see the Software Engineering Models and Methods KA [1*, c17s2]).
- Testing vs. Debugging (see Construction Testing in the Software Construction KA and Debugging Tools and Techniques in the Computing Foundations KA [1*, c3s6]).

- Testing vs. Program Construction (see Construction Testing in the Software Construction KA [1*, c3s2]).

2. Test Levels

Software testing is usually performed at different *levels* throughout the development and maintenance processes. Levels can be distinguished based on the object of testing, which is called the *target*, or on the purpose, which is called the *objective* (of the test level).

2.1. The Target of the Test

[1*, c1s13] [2*, c8s1]

The target of the test can vary: a single module, a group of such modules (related by purpose, use, behavior, or structure), or an entire system. Three test stages can be distinguished: unit, integration, and system. These three test stages do not imply any process model, nor is any one of them assumed to be more important than the other two.

2.1.1. Unit Testing

[1*, c3] [2*, c8]

Unit testing verifies the functioning in isolation of software elements that are separately testable. Depending on the context, these could be the individual subprograms or a larger component made of highly cohesive units. Typically, unit testing occurs with access to the code being tested and with the support of debugging tools. The programmers who wrote the code typically, but not always, conduct unit testing.

2.1.2. Integration Testing

[1*, c7] [2*, c8]

Integration testing is the process of verifying the interactions among software components. Classical integration testing strategies, such as top-down and bottom-up, are often used with hierarchically structured software.

Modern, systematic integration strategies are typically architecture-driven, which involves incrementally integrating the software components or subsystems based on identified functional threads. Integration testing is often an ongoing activity at each stage of development during which software engineers abstract away lower-level perspectives and concentrate on the perspectives of the level at which they are integrating. For other than small, simple software, incremental integration testing strategies are usually preferred to putting all of the components together at once—which is often called "big bang" testing.

2.1.3. System Testing

[1*, c8] [2*, c8]

System testing is concerned with testing the behavior of an entire system. Effective unit and integration testing will have identified many of the software defects. System testing is usually considered appropriate for assessing the nonfunctional system requirements—such as security, speed, accuracy, and reliability (see Functional and Non-Functional Requirements in the Software Requirements KA and Software Quality Requirements in the Software Quality KA). External interfaces to other applications, utilities, hardware devices, or the operating environments are also usually evaluated at this level.

2.2. Objectives of Testing

[1*, c1s7]

Testing is conducted in view of specific objectives, which are stated more or less explicitly and with varying degrees of precision. Stating the objectives of testing in precise, quantitative terms supports measurement and control of the test process.

Testing can be aimed at verifying different properties. Test cases can be designed to check that the functional specifications are correctly implemented, which is variously referred to in the literature as conformance testing, correctness testing, or functional testing. However, several other nonfunctional properties may be tested as well—including performance, reliability, and usability, among many others (see Models and Quality Characteristics in the Software Quality KA).

Other important objectives for testing include but are not limited to reliability measurement,

identification of security vulnerabilities, usability evaluation, and software acceptance, for which different approaches would be taken. Note that, in general, the test objectives vary with the test target; different purposes are addressed at different levels of testing.

The subtopics listed below are those most often cited in the literature. Note that some kinds of testing are more appropriate for custom-made software packages—installation testing, for example—and others for consumer products, like beta testing.

2.2.1. Acceptance / Qualification Testing

[1*, c1s7] [2*, c8s4]

Acceptance / qualification testing determines whether a system satisfies its acceptance criteria, usually by checking desired system behaviors against the customer's requirements. The customer or a customer's representative thus specifies or directly undertakes activities to check that their requirements have been met, or in the case of a consumer product, that the organization has satisfied the stated requirements for the target market. This testing activity may or may not involve the developers of the system.

2.2.2. Installation Testing

[1*, c12s2]

Often, after completion of system and acceptance testing, the software is verified upon installation in the target environment. Installation testing can be viewed as system testing conducted in the operational environment of hardware configurations and other operational constraints. Installation procedures may also be verified.

2.2.3. Alpha and Beta Testing

[1*, c13s7, c16s6] [2*, c8s4]

Before software is released, it is sometimes given to a small, selected group of potential users for trial use (*alpha* testing) and/or to a larger set of representative users (*beta* testing). These users report problems with the product. Alpha and beta testing are often uncontrolled and are not always referred to in a test plan.

2.2.4. Reliability Achievement and Evaluation

[1*, c15] [2*, c15s2]

Testing improves reliability by identifying and correcting faults. In addition, statistical measures of reliability can be derived by randomly generating test cases according to the operational profile of the software (see Operational Profile in section 3.5, Usage-Based Techniques). The latter approach is called *operational testing*. Using reliability growth models, both objectives can be pursued together [3] (see *L*ife Test, Reliability Evaluation in section 4.1, Evaluation of the Program under Test).

2.2.5. Regression Testing

[1*, c8s11, c13s3]

According to [7], regression testing is the "selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements." In practice, the approach is to show that software still passes previously passed tests in a test suite (in fact, it is also sometimes referred to as nonregression testing). For incremental development, the purpose of regression testing is to show that software behavior is unchanged by incremental changes to the software, except insofar as it should. In some cases, a tradeoff must be made between the assurance given by regression testing every time a change is made and the resources required to perform the regression tests, which can be quite time consuming due to the large number of tests that may be executed. Regression testing involves selecting, minimizing, and/or prioritizing a subset of the test cases in an existing test suite [8]. Regression testing can be conducted at each of the test levels described in section 2.1, The Target of the Test, and may apply to functional and nonfunctional testing.

2.2.6. Performance Testing

[1*, c8s6]

Performance testing verifies that the software meets the specified performance requirements and assesses performance characteristics—for instance, capacity and response time.

2.2.7. Security Testing

[1*, c8s3] [2*, c11s4]

Security testing is focused on the verification that the software is protected from external attacks. In particular, security testing verifies the confidentiality, integrity, and availability of the systems and its data. Usually, security testing includes verification against misuse and abuse of the software or system (negative testing).

2.2.8. Stress Testing

[1*, c8s8]

Stress testing exercises software at the maximum design load, as well as beyond it, with the goal of determining the behavioral limits, and to test defense mechanisms in critical systems.

2.2.9. Back-to-Back Testing

[7]

IEEE/ISO/IEC Standard 24765 defines back-to-back testing as "testing in which two or more variants of a program are executed with the same inputs, the outputs are compared, and errors are analyzed in case of discrepancies."

2.2.10. Recovery Testing

[1*, c14s2]

Recovery testing is aimed at verifying software restart capabilities after a system crash or other "disaster."

2.2.11. Interface Testing

[2*, c8s1.3] [9*, c4s4.5]

Interface defects are common in complex systems. Interface testing aims at verifying whether the components interface correctly to provide the correct exchange of data and control information. Usually the test cases are generated from the interface specification. A specific objective of interface testing is to simulate the use of APIs by end-user applications. This involves the generation of parameters of the API calls, the setting of external environment conditions, and the definition of internal data that affect the API.

2.2.12. Configuration Testing

[1*, c8s5]

In cases where software is built to serve different users, configuration testing verifies the software under different specified configurations.

2.2.13. Usability and Human Computer Interaction Testing

[10*, c6]

The main task of usability and human computer interaction testing is to evaluate how easy it is for end users to learn and to use the software. In general, it may involve testing the software functions that supports user tasks, documentation that aids users, and the ability of the system to recover from user errors (see User Interface Design in the Software Design KA).

3. Test Techniques

One of the aims of testing is to detect as many failures as possible. Many techniques have been developed to do this [6, part 4]. These techniques attempt to "break" a program by being as systematic as possible in identifying inputs that will produce representative program behaviors; for instance, by considering subclasses of the input domain, scenarios, states, and data flows.

The classification of testing techniques presented here is based on how tests are generated: from the software engineer's intuition and experience, the specifications, the code structure, the real or imagined faults to be discovered, predicted usage, models, or the nature of the application. One category deals with the combined use of two or more techniques.

Sometimes these techniques are classified as *white-box* (also called *glass-box*), if the tests are based on information about how the software has been designed or coded, or as *black-box* if the test cases rely only on the input/output behavior of the software. The following list includes those testing techniques that are commonly used, but some practitioners rely on some of the techniques more than others.

3.1. Based on the Software Engineer's Intuition and Experience

3.1.1. Ad Hoc

Perhaps the most widely practiced technique is ad hoc testing: tests are derived relying on the software engineer's skill, intuition, and experience with similar programs. Ad hoc testing can be useful for identifying tests cases that not easily generated by more formalized techniques.

3.1.2. Exploratory Testing

Exploratory testing is defined as simultaneous learning, test design, and test execution [6, part 1]; that is, the tests are not defined in advance in an established test plan, but are dynamically designed, executed, and modified. The effectiveness of exploratory testing relies on the software engineer's knowledge, which can be derived from various sources: observed product behavior during testing, familiarity with the application, the platform, the failure process, the type of possible faults and failures, the risk associated with a particular product, and so on.

3.2. Input Domain-Based Techniques

3.2.1. Equivalence Partitioning

[1*, c9s4]

Equivalence partitioning involves partitioning the input domain into a collection of subsets (or equivalent classes) based on a specified criterion or relation. This criterion or relation may be different computational results, a relation based on control flow or data flow, or a distinction made between valid inputs that are accepted and processed by the system and invalid inputs, such as out of range values, that are not accepted and should generate an error message or initiate error processing. A representative set of tests (sometimes only one) is usually taken from each equivalency class.

3.2.2. Pairwise Testing

[1*, c9s3]

Test cases are derived by combining interesting values for every pair of a set of input variables instead of considering all possible combinations. Pairwise testing belongs to combinatorial testing, which in general also includes higher-level combinations than pairs: these techniques are referred to as *t-wise*, whereby every possible combination of *t* input variables is considered.

3.2.3. Boundary-Value Analysis

[1*, c9s5]

Test cases are chosen on or near the boundaries of the input domain of variables, with the underlying rationale that many faults tend to concentrate near the extreme values of inputs. An extension of this technique is robustness testing, wherein test cases are also chosen outside the input domain of variables to test program robustness in processing unexpected or erroneous inputs.

3.2.4. Random Testing

[1*, c9s7]

Tests are generated purely at random (not to be confused with statistical testing from the operational profile, as described in Operational Profile in section 3.5). This form of testing falls under the heading of input domain testing since the input domain must be known in order to be able to pick random points within it. Random testing provides a relatively simple approach for test automation; recently, enhanced forms of random testing have been proposed in which the random input sampling is directed by other input selection criteria [11]. Fuzz testing or fuzzing is a special form of random testing aimed at breaking the software; it is most often used for security testing.

3.3. Code-Based Techniques

3.3.1. Control Flow-Based Criteria

[1*, c4]

Control flow-based coverage criteria are aimed at covering all the statements, blocks of statements, or specified combinations of statements in a program. The strongest of the control flow-based criteria is path testing, which aims to execute all entry-to-exit control flow paths in a program's control flow graph. Since exhaustive path testing is generally not feasible because of

loops, other less stringent criteria focus on coverage of paths that limit loop iterations such as statement coverage, branch coverage, and condition/decision testing. The adequacy of such tests is measured in percentages; for example, when all branches have been executed at least once by the tests, 100% branch coverage has been achieved.

3.3.2. Data Flow-Based Criteria

[1*, c5]

In data flow-based testing, the control flow graph is annotated with information about how the program variables are defined, used, and killed (undefined). The strongest criterion, all definition-use paths, requires that, for each variable, every control flow path segment from a definition of that variable to a use of that definition is executed. In order to reduce the number of paths required, weaker strategies such as all-definitions and all-uses are employed.

3.3.3. Reference Models for Code-Based Testing

[1*, c4]

Although not a technique in itself, the control structure of a program can be graphically represented using a flow graph to visualize code-based testing techniques. A flow graph is a directed graph, the nodes and arcs of which correspond to program elements (see Graphs and Trees in the Mathematical Foundations KA). For instance, nodes may represent statements or uninterrupted sequences of statements, and arcs may represent the transfer of control between nodes.

3.4. Fault-Based Techniques

[1*, c1s14]

With different degrees of formalization, fault-based testing techniques devise test cases specifically aimed at revealing categories of likely or predefined faults. To better focus the test case generation or selection, a *fault model* can be introduced that classifies the different types of faults.

3.4.1. Error Guessing

[1*, c9s8]

In error guessing, test cases are specifically designed by software engineers who try to anticipate the most plausible faults in a given program. A good source of information is the history of faults discovered in earlier projects, as well as the software engineer's expertise.

3.4.2. Mutation Testing

[1*, c3s5]

A mutant is a slightly modified version of the program under test, differing from it by a small syntactic change. Every test case exercises both the original program and all generated mutants: if a test case is successful in identifying the difference between the program and a mutant, the latter is said to be "killed." Originally conceived as a technique to evaluate test sets (see section 4.2. Evaluation of the Tests Performed), mutation testing is also a testing criterion in itself: either tests are randomly generated until enough mutants have been killed, or tests are specifically designed to kill surviving mutants. In the latter case, mutation testing can also be categorized as a code-based technique. The underlying assumption of mutation testing, the coupling effect, is that by looking for simple syntactic faults, more complex but real faults will be found. For the technique to be effective, a large number of mutants must be automatically generated and executed in a systematic way [12].

3.5. Usage-Based Techniques

3.5.1. Operational Profile

[1*, c15s5]

In testing for reliability evaluation (also called operational testing), the test environment reproduces the operational environment of the software, or the *operational profile*, as closely as possible. The goal is to infer from the observed test results the future reliability of the software when in actual use. To do this, inputs are assigned probabilities, or profiles, according to their frequency of occurrence in actual operation. Operational profiles can be used during system testing

to guide derivation of test cases that will assess the achievement of reliability objectives and exercise relative usage and criticality of different functions similar to what will be encountered in the operational environment [3].

3.5.2. User Observation Heuristics

[10*, c5, c7]

Usability principles can provide guidelines for discovering problems in the design of the user interface [10*, c1s4] (see User Interface Design in the Software Design KA). Specialized heuristics, also called usability inspection methods, are applied for the systematic observation of system usage under controlled conditions in order to determine how well people can use the system and its interfaces. Usability heuristics include cognitive walkthroughs, claims analysis, field observations, thinking aloud, and even indirect approaches such as user questionnaires and interviews.

3.6. Model-Based Testing Techniques

A model in this context is an abstract (formal) representation of the software under test or of its software requirements (see Modeling in the Software Engineering Models and Methods KA). Model-based testing is used to validate requirements, check their consistency, and generate test cases focused on the behavioral aspects of the software. The key components of model-based testing are [13]: the notation used to represent the model of the software or its requirements; workflow models or similar models; the test strategy or algorithm used for test case generation; the supporting infrastructure for the test execution; and the evaluation of test results compared to expected results. Due to the complexity of the techniques, model-based testing approaches are often used in conjunction with test automation harnesses. Model-based testing techniques include the following.

3.6.1. Decision Tables

[1*, c9s6]

Decision tables represent logical relationships between conditions (roughly, inputs) and actions (roughly, outputs). Test cases are systematically derived by considering every possible combination of conditions and their corresponding resultant actions. A related technique is *cause-effect graphing* [1*, c13s6].

3.6.2. Finite-State Machines

[1*, c10]

By modeling a program as a finite state machine, tests can be selected in order to cover the states and transitions.

3.6.3. Formal Specifications

[1*, c10s11] [2*, c15]

Stating the specifications in a formal language (see Formal Methods in the Software Engineering Models and Methods KA) permits automatic derivation of functional test cases, and, at the same time, provides an oracle for checking test results.

TTCN3 (Testing and Test Control Notation version 3) is a language developed for writing test cases. The notation was conceived for the specific needs of testing telecommunication systems, so it is particularly suitable for testing complex communication protocols.

3.6.4. Workflow Models

[2*, c8s3.2, c19s3.1]

Workflow models specify a sequence of activities performed by humans and/or software applications, usually represented through graphical notations. Each sequence of actions constitutes one workflow (also called a scenario). Both typical and alternate workflows should be tested [6, part 4]. A special focus on the roles in a workflow specification is targeted in business process testing.

3.7. Techniques Based on the Nature of the Application

The above techniques apply to all kinds of software. Additional techniques for test derivation and execution are based on the nature of the software being tested; for example,

- object-oriented software
- component-based software
- web-based software
- concurrent programs
- protocol-based software
- real-time systems
- safety-critical systems
- service-oriented software
- open-source software
- embedded software

3.8. Selecting and Combining Techniques

3.8.1. Combining Functional and Structural

[1*, c9]

Model-based and code-based test techniques are often contrasted as functional vs. structural testing. These two approaches to test selection are not to be seen as alternatives but rather as complements; in fact, they use different sources of information and have been shown to highlight different kinds of problems. They could be used in combination, depending on budgetary considerations.

3.8.2. Deterministic vs. Random

[1*, c9s6]

Test cases can be selected in a deterministic way, according to one of many techniques, or randomly drawn from some distribution of inputs, such as is usually done in reliability testing. Several analytical and empirical comparisons have been conducted to analyze the conditions that make one approach more effective than the other.

4. Test-Related Measures

Sometimes testing techniques are confused with testing objectives. Testing techniques can be viewed as aids that help to ensure the achievement of test objectives [6, part 4]. For instance, branch coverage is a popular testing technique. Achieving a specified branch coverage measure (e.g., 95% branch coverage) should not be the objective of testing per se: it is a way of improving the chances of finding failures by attempting to systematically exercise every program branch at every decision point. To avoid such misunderstandings, a clear distinction should be made between test-related measures that provide an evaluation of the program under test, based on the observed test outputs, and the measures that evaluate the thoroughness of the test set. (See Software Engineering Measurement in the Software Engineering Management KA for information on measurement programs. See Software Process and Product Measurement in the Software Engineering Process KA for information on measures.)

Measurement is usually considered fundamental to quality analysis. Measurement may also be used to optimize the planning and execution of the tests. Test management can use several different process measures to monitor progress. (See section 5.1, Practical Considerations, for a discussion of measures of the testing process useful for management purposes.)

4.1. Evaluation of the Program Under Test

4.1.1. Program Measurements That Aid in Planning and Designing Tests

[9*, c11]

Measures based on software size (for example, source lines of code or functional size; see Measuring Requirements in the Software Requirements KA) or on program structure can be used to guide testing. Structural measures also include measurements that determine the frequency with which modules call one another.

4.1.2. Fault Types, Classification, and Statistics

[9*, c4]

The testing literature is rich in classifications and taxonomies of faults. To make testing more effective, it is important to know which types of faults may be found in the software under test and the relative frequency with which these faults have occurred in the past. This information can be useful in making quality predictions as well as in process improvement (see Defect Characterization in the Software Quality KA).

4.1.3. Fault Density

[1*, c13s4] [9*, c4]

A program under test can be evaluated by counting discovered faults as the ratio between the number of faults found and the size of the program.

4.1.4. Life Test, Reliability Evaluation

[1*, c15] [9*, c3]

A statistical estimate of software reliability, which can be obtained by observing reliability achieved, can be used to evaluate a software product and decide whether or not testing can be stopped (see section 2.2, Reliability Achievement and Evaluation).

4.1.5. Reliability Growth Models

[1*, c15] [9*, c8]

Reliability growth models provide a prediction of reliability based on failures. They assume, in general, that when the faults that caused the observed failures have been fixed (although some models also accept imperfect fixes), the estimated product's reliability exhibits, on average, an increasing trend. There are many published reliability growth models. Notably, these models are divided into *failure-count* and *time-between-failure* models.

4.2. Evaluation of the Tests Performed

4.2.1. Coverage / Thoroughness Measures

[9*, c11]

Several test adequacy criteria require that the test cases systematically exercise a set of elements identified in the program or in the specifications (see topic 3, Test Techniques). To evaluate the thoroughness of the executed tests, software engineers can monitor the elements covered so that they can dynamically measure the ratio between covered elements and the total number. For example, it is possible to measure the percentage of branches covered in the program flow graph or the percentage of functional requirements exercised among those listed in the specifications document. Code-based adequacy criteria require appropriate instrumentation of the program under test.

4.2.2. Fault Seeding

[1*, c2s5] [9*, c6]

In fault seeding, some faults are artificially introduced into a program before testing. When the tests are executed, some of these seeded faults will be revealed as well as, possibly, some faults that were already there. In theory, depending on which and how many of the artificial faults are discovered, testing effectiveness can be evaluated and the remaining number of genuine faults can be estimated. In practice, statisticians question the distribution and representativeness of seeded faults relative to genuine faults and the small sample size on which any extrapolations are based. Some also argue that this technique should be used with great care since inserting faults into software involves the obvious risk of leaving them there.

4.2.3. Mutation Score

[1*, c3s5]

In mutation testing (see Mutation Testing in section 3.4, Fault-Based Techniques), the ratio of killed mutants to the total number of generated mutants can be a measure of the effectiveness of the executed test set.

4.2.4. Comparison and Relative Effectiveness of Different Techniques

Several studies have been conducted to compare the relative effectiveness of different testing techniques. It is important to be precise as to the property against which the techniques are being assessed; what, for instance, is the exact meaning given to the term "effectiveness"? Possible interpretations include the number of tests needed to find the first failure, the ratio of the number of faults found through testing to all the faults found during and after testing, and how much reliability was improved. Analytical and empirical comparisons between different techniques have been conducted according to each of the notions of effectiveness specified above.

5. Test Process

Testing concepts, strategies, techniques, and measures need to be integrated into a defined and

controlled process. The test process supports testing activities and provides guidance to testers and testing teams, from test planning to test output evaluation, in such a way as to provide assurance that the test objectives will be met in a cost-effective way.

5.1. Practical Considerations

5.1.1. Attitudes / Egoless Programming

[1*c16] [9*, c15]

An important element of successful testing is a collaborative attitude towards testing and quality assurance activities. Managers have a key role in fostering a generally favorable reception towards failure discovery and correction during software development and maintenance; for instance, by overcoming the mindset of individual code ownership among programmers and by promoting a collaborative environment with team responsibility for anomalies in the code.

5.1.2. Test Guides

[1*, c12s1] [9*, c15s1]

The testing phases can be guided by various aims—for example, risk-based testing uses the product risks to prioritize and focus the test strategy, and scenario-based testing defines test cases based on specified software scenarios.

5.1.3. Test Process Management

[1*, c12] [9*, c15]

Test activities conducted at different levels (see topic 2, Test Levels) must be organized—together with people, tools, policies, and measures—into a well-defined process that is an integral part of the life cycle.

5.1.4. Test Documentation and Work Products

[1*, c8s12] [9*, c4s5]

Documentation is an integral part of the formalization of the test process [6, part 3]. Test documents may include, among others, the test plan, test design specification, test procedure specification, test case specification, test log, and test incident report. The software under test is documented as the test item. Test documentation should be produced and continually updated to the same level of quality as other types of documentation in software engineering. Test documentation should also be under the control of software configuration management (see the Software Configuration Management KA). Moreover, test documentation includes work products that can provide material for user manuals and user training.

5.1.5. Test-Driven Development

[1*, c1s16]

Test-driven development (TDD) originated as one of the core XP (extreme programming) practices and consists of writing unit tests prior to writing the code to be tested (see Agile Methods in the Software Engineering Models and Method KA). In this way, TDD develops the test cases as a surrogate for a software requirements specification document rather than as an independent check that the software has correctly implemented the requirements. Rather than a testing strategy, TDD is a practice that requires software developers to define and maintain unit tests; it thus can also have a positive impact on elaborating user needs and software requirements specifications.

5.1.6. Internal vs. Independent Test Team

[1*, c16]

Formalizing the testing process may also involve formalizing the organization of the testing team. The testing team can be composed of internal members (that is, on the project team, involved or not in software construction), of external members (in the hope of bringing an unbiased, independent perspective), or of both internal and external members. Considerations of cost, schedule, maturity levels of the involved organizations, and criticality of the application can guide the decision.

5.1.7. Cost/Effort Estimation and Test Process Measures

[1*, c18s3] [9*, c5s7]

Several measures related to the resources spent on testing, as well as to the relative fault-finding effectiveness of the various test phases, are used by managers to control and improve the testing

process. These test measures may cover such aspects as number of test cases specified, number of test cases executed, number of test cases passed, and number of test cases failed, among others.

Evaluation of test phase reports can be combined with root-cause analysis to evaluate test-process effectiveness in finding faults as early as possible. Such an evaluation can be associated with the analysis of risks. Moreover, the resources that are worth spending on testing should be commensurate with the use/criticality of the application: different techniques have different costs and yield different levels of confidence in product reliability.

5.1.8. Termination

[9*, c10s4]

A decision must be made as to how much testing is enough and when a test stage can be terminated. Thoroughness measures, such as achieved code coverage or functional coverage, as well as estimates of fault density or of operational reliability, provide useful support but are not sufficient in themselves. The decision also involves considerations about the costs and risks incurred by possible remaining failures, as opposed to the costs incurred by continuing to test (see Test Selection Criteria / Test Adequacy Criteria in section 1.2, Key Issues).

5.1.9. Test Reuse and Test Patterns

[9*, c2s5]

To carry out testing or maintenance in an organized and cost-effective way, the means used to test each part of the software should be reused systematically. A repository of test materials should be under the control of software configuration management so that changes to software requirements or design can be reflected in changes to the tests conducted.

The test solutions adopted for testing some application types under certain circumstances, with the motivations behind the decisions taken, form a test pattern that can itself be documented for later reuse in similar projects.

5.2. Test Activities

As shown in the following description, successful management of test activities strongly depends on the software configuration management process (see the Software Configuration Management KA).

5.2.1. Planning

[1*, c12s1, c12s8]

Like all other aspects of project management, testing activities must be planned. Key aspects of test planning include coordination of personnel, availability of test facilities and equipment, creation and maintenance of all test-related documentation, and planning for possible undesirable outcomes. If more than one baseline of the software is being maintained, then a major planning consideration is the time and effort needed to ensure that the test environment is set to the proper configuration.

5.2.2. Test-Case Generation

[1*, c12s1, c12s3]

Generation of test cases is based on the level of testing to be performed and the particular testing techniques. Test cases should be under the control of software configuration management and include the expected results for each test.

5.2.3. Test Environment Development

[1*, c12s6]

The environment used for testing should be compatible with the other adopted software engineering tools. It should facilitate development and control of test cases, as well as logging and recovery of expected results, scripts, and other testing materials.

5.2.4. Execution

[1*, c12s7]

Execution of tests should embody a basic principle of scientific experimentation: everything done during testing should be performed and documented clearly enough that another person

could replicate the results. Hence, testing should be performed in accordance with documented procedures using a clearly defined version of the software under test.

5.2.5. Test Results Evaluation

[9*, c15]

The results of testing should be evaluated to determine whether or not the testing has been successful. In most cases, "successful" means that the software performed as expected and did not have any major unexpected outcomes. Not all unexpected outcomes are necessarily faults but are sometime determined to be simply noise. Before a fault can be removed, an analysis and debugging effort is needed to isolate, identify, and describe it. When test results are particularly important, a formal review board may be convened to evaluate them.

5.2.6. Problem Reporting / Test Log

[1*, c13s9]

Testing activities can be entered into a testing log to identify when a test was conducted, who performed the test, what software configuration was used, and other relevant identification information. Unexpected or incorrect test results can be recorded in a problem reporting system, the data for which forms the basis for later debugging and fixing the problems that were observed as failures during testing. Also, anomalies not classified as faults could be documented in case they later turn out to be more serious than first thought. Test reports are also inputs to the change management request process (see Software Configuration Control in the Software Configuration Management KA).

5.2.7. Defect Tracking

[9*, c9]

Defects can be tracked and analyzed to determine when they were introduced into the software, why they were created (for example, poorly defined requirements, incorrect variable declaration, memory leak, programming syntax error), and when they could have been first observed in the software. Defect tracking information is used to determine what aspects of software testing and other processes need improvement and how effective previous approaches have been.

6. Software Testing Tools

6.1. Testing Tool Support

[1*, c12s11] [9*, c5]

Testing requires many labor-intensive tasks, running numerous program executions, and handling a great amount of information. Appropriate tools can alleviate the burden of clerical, tedious operations and make them less error-prone. Sophisticated tools can support test design and test case generation, making it more effective.

6.1.1. Selecting Tools

[1*, c12s11]

Guidance to managers and testers on how to select testing tools that will be most useful to their organization and processes is a very important topic, as tool selection greatly affects testing efficiency and effectiveness. Tool selection depends on diverse evidence, such as development choices, evaluation objectives, execution facilities, and so on. In general, there may not be a unique tool that will satisfy particular needs, so a suite of tools could be an appropriate choice.

6.2. Categories of Tools

We categorize the available tools according to their functionality:

- *Test harnesses* (drivers, stubs) [1*, c3s9] provide a controlled environment in which tests can be launched and the test outputs can be logged. In order to execute parts of a program, drivers and stubs are provided to simulate calling and called modules, respectively.
- *Test generators* [1*, c12s11] provide assistance in the generation test cases. The generation can be random, path-based, model-based, or a mix thereof.
- *Capture/replay tools* [1*, c12s11] automatically reexecute, or replay, previously

executed tests which have recorded inputs and outputs (e.g., screens).

- *Oracle/file comparators/assertion checking tools* [1*, c9s7] assist in deciding whether a test outcome is successful or not.
- *Coverage analyzers and instrumenters* [1*, c4] work together. Coverage analyzers assess which and how many entities of the program flow graph have been exercised amongst all those required by the selected test coverage criterion. The analysis can be done thanks to program instrumenters that insert recording probes into the code.
- *Tracers* [1*, c1s7] record the history of a program's execution paths.
- *Regression testing tools* [1*, c12s16] support the reexecution of a test suite after a section of software has been modified. They can also help to select a test subset according to the change made.
- *Reliability evaluation tools* [9*, c8] support test results analysis and graphical visualization in order to assess reliability-related measures according to selected models.

MATRIX OF TOPICS VS. REFERENCE MATERIAL

	Naik and Tripathy 2008 [1*]	Sommerville 2011 [2*]	Kan 2003 [9*]	Nielsen 1993 [10*]
1. Software Testing Fundamentals				
1.1. Testing-Related Terminology				
1.1.1. Definitions of Testing and Related Terminology	c1,c2	c8		
1.1.2. Faults vs. Failures	c1s5	c11		
1.2. Key Issues				
1.2.1. Test Selection Criteria / Test Adequacy Criteria (Stopping Rules)	c1s14, c6s6, c12s7			
1.2.2. Testing Effectiveness / Objectives for Testing	c13s11, c11s4			
1.2.3. Testing for Defect Identification	c1s14			
1.2.4. The Oracle Problem	c1s9, c9s7			
1.2.5. Theoretical and Practical Limitations of Testing	c2s7			
1.2.6. The Problem of Infeasible Paths	c4s7			
1.2.7. Testability	c17s2			
1.3. Relationship of Testing to Other Activities				
1.3.1. Testing vs. Static Software Quality Management Techniques	c12			
1.3.2. Testing vs. Correctness Proofs and Formal Verification	c17s2			
1.3.3. Testing vs. Debugging	c3s6			
1.3.4. Testing vs. Programming	c3s2			
2. Test Levels				
2.1. The Target of the Test	c1s13	c8s1		
2.1.1. Unit Testing	c3	c8		
2.1.2. Integration Testing	c7	c8		
2.1.3. System Testing	c8	c8		

	Naik and Tripathy 2008 [1*]	Sommerville 2011 [2*]	Kan 2003 [9*]	Nielsen 1993 [10*]
2.2. Objectives of Testing	c1s7			
2.2.1. Acceptance / Qualification	c1s7	c8s4		
2.2.2. Installation Testing	c12s2			
2.2.3. Alpha and Beta Testing	c13s7, c16s6	c8s4		
2.2.4. Reliability Achievement and Evaluation	c15	c15s2		
2.2.5. Regression Testing	c8s11, c13s3			
2.2.6. Performance Testing	c8s6			
2.2.7. Security Testing	c8s3	c11s4		
2.2.8. Stress Testing	c8s8			
2.2.9. Back-to-Back Testing				
2.2.10. Recovery Testing	c14s2			
2.2.11. Interface Testing		c8s1.3	c4s4.5	
2.2.12. Configuration Testing	c8s5			
2.2.13. Usability and Human Computer Interaction Testing				c6
3. Test Techniques				
3.1. Based on the Software Engineer's Intuition and Experience				
3.1.1. Ad Hoc				
3.1.2. Exploratory Testing				
3.2. Input Domain-Based Techniques				
3.2.1. Equivalence Partitioning	c9s4			
3.2.2. Pairwise Testing	c9s3			
3.2.3. Boundary-Value Analysis	c9s5			
3.2.4. Random Testing	c9s7			
3.3. Code-Based Techniques				
3.3.1. Control Flow-Based Criteria	c4			

	Naik and Tripathy 2008 [1*]	Sommerville 2011 [2*]	Kan 2003 [9*]	Nielsen 1993 [10*]
3.3.2. Data Flow-Based Criteria	c5			
3.3.3. Reference Models for Code-Based Testing	c4			
3.4. Fault-Based Techniques	c1s14			
3.4.1. Error Guessing	c9s8			
3.4.2. Mutation Testing	c3s5			
3.5. Usage-Based Techniques				
3.5.1. Operational Profile	c15s5			
3.5.2. User Observation Heuristics				c5, c7
3.6. Model-Based Testing Techniques				
3.6.1. Decision Table	c9s6			
3.6.2. Finite-State Machines	c10			
3.6.3. Testing from Formal Specifications	c10s11	c15		
3.7. Techniques Based on the Nature of the Application				
3.8. Selecting and Combining Techniques				
3.8.1. Functional and Structural	c9			
3.8.2. Deterministic vs. Random	c9s6			
4. Test-Related Measures				
4.1. Evaluation of the Program Under Test				
4.1.1. Program Measurements That Aid in Planning and Designing Testing			c11	
4.1.2. Fault Types, Classification, and Statistics			c4	
4.1.3. Fault Density	c13s4		c4	
4.1.4. Life Test, Reliability Evaluation	c15		c3	
4.1.5. Reliability Growth Models	c15		c8	

	Naik and Tripathy 2008 [1*]	Sommerville 2011 [2*]	Kan 2003 [9*]	Nielsen 1993 [10*]
4.2. Evaluation of the Tests Performed				
4.2.1. Coverage / Thoroughness Measures			c11	
4.2.2. Fault Seeding	c2s5		c6	
4.2.3. Mutation Score	c3s5			
4.2.4. Comparison and Relative Effectiveness of Different Techniques				
5. Test Process				
5.1. Practical Considerations				
5.1.1. Attitudes / Egoless Programming	c16		c15	
5.1.2. Test Guides	c12s1		c15s1	
5.1.3. Test Process Management	c12		c15	
5.1.4. Test Documentation and Work Products	c8s12		c4s5	
5.1.5. Test-Driven Development	c1s16			
5.1.6. Internal vs. Independent Test Team	c16			
5.1.7. Cost/Effort Estimation and Other Process Measures	c18s3		c5s7	
5.1.8. Termination			c10s4	
5.1.9. Test Reuse and Patterns			c2s5	
5.2. Test Activities				
5.2.1. Planning	c12s1 c12s8			
5.2.2. Test-Case Generation	c12s1 c12s3			
5.2.3. Test Environment Development	c12s6			
5.2.4. Execution	c12s7			
5.2.5. Test Results Evaluation			c15	

	Naik and Tripathy 2008 [1*]	Sommerville 2011 [2*]	Kan 2003 [9*]	Nielsen 1993 [10*]
5.2.6. Problem Reporting / Test Log	c13s9			
5.2.7. Defect Tracking			c9	
6. Software Testing Tools				
6.1. Testing Tool Support	c12s11		c5	
6.1.1. Selecting Tools	c12s11			
6.2. Categories of Tools	c1s7, c3s9, c4, c9s7, c12s11, c12s16		c8	